NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

INVESTIGATING THE FEASIBILITY OF CONDUCTING HUMAN TRACKING AND FOLLOWING IN AN INDOOR ENVIRONMENT USING A MICROSOFT KINECT AND THE ROBOT OPERATING SYSTEM

by

Rebecca A. Greenberg

June 2017

Thesis Advisor:	Xiaoping Yun
Co-Advisor:	James Calusdian

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY	**2. REPORT DATE** June 2017	**3. REPORT TYPE AND DATES COVERED** Master's thesis
4. TITLE AND SUBTITLE INVESTIGATING THE FEASIBILITY OF CONDUCTING HUMAN TRACKING AND FOLLOWING IN AN INDOOR ENVIRONMENT USING A MICROSOFT KINECT AND THE ROBOT OPERATING SYSTEM		**5. FUNDING NUMBERS**
6. AUTHOR(S) Rebecca A. Greenberg		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Human detection, tracking, and following is one application in which computer vision can be relevant to robotics. By using a sequence of images, a human can be found and that human's movement can be followed. The Microsoft Kinect, one of the most successful color image and depth (RGB-D) sensors, is known for its human detection capabilities and has multiple software development kits available. The objective of this thesis was to determine if it was feasible to implement human tracking and following on a mobile robot in an indoor environment. Specifically, the tracking was conducted with the Microsoft Kinect and a specific software development environment, Robot Operating System (ROS) and MATLAB. ROS was utilized to run the drivers for the robot and the Microsoft Kinect, while MATLAB was utilized to run the algorithms and experiments. The skeleton tracking capabilities of the Kinect were utilized as the main tracking system. An auxiliary method was created by using histograms of depth and region properties to segment a person from a depth image. The indoor robot was able to successfully track and follow a person through the indoor environment using the raw sensor data and a combination of the two tracking methods.

14. SUBJECT TERMS robot, Robot Operating System, Kinect, P3-DX, human detection, human tracking and following			**15. NUMBER OF PAGES** 121
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

INVESTIGATING THE FEASIBILITY OF CONDUCTING HUMAN TRACKING AND FOLLOWING IN AN INDOOR ENVIRONMENT USING A MICROSOFT KINECT AND THE ROBOT OPERATING SYSTEM

Rebecca A. Greenberg
Ensign, United States Navy
B.S., U.S. Naval Academy, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2017

Approved by: Xiaoping Yun
Thesis Advisor

James Calusdian
Co-Advisor

R. Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Human detection, tracking, and following is one application in which computer vision can be relevant to robotics. By using a sequence of images, a human can be found and that human's movement can be followed. The Microsoft Kinect, one of the most successful color image and depth (RGB-D) sensors, is known for its human detection capabilities and has multiple software development kits available. The objective of this thesis was to determine if it was feasible to implement human tracking and following on a mobile robot in an indoor environment. Specifically, the tracking was conducted with the Microsoft Kinect and a specific software development environment, Robot Operating System (ROS) and MATLAB. ROS was utilized to run the drivers for the robot and the Microsoft Kinect, while MATLAB was utilized to run the algorithms and experiments. The skeleton tracking capabilities of the Kinect were utilized as the main tracking system. An auxiliary method was created by using histograms of depth and region properties to segment a person from a depth image. The indoor robot was able to successfully track and follow a person through the indoor environment using the raw sensor data and a combination of the two tracking methods.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

HOD	histogram of oriented depth
HOG	histogram of oriented gradients
IP	Internet protocol
IR	infrared
LS3	Legged Squad Support System
P3-DX	Pioneer 3 differential drive mobile robot
RGB-D	red, green, blue and depth
ROS	Robot Operating System
SDK	software development kit
SLAM	simultaneous localization and mapping
SSH	secure shell
URDF	unified robot description format

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First, I would like to thank my advisors, Professor Xiaoping Yun and Professor James Calusdian, for helping me through the thesis process. Professor Yun, thank you for the introduction to graduate-level robotics and for helping me find a project related to my interest. Professor Calusdian, thank you for all of the support you provide within the lab. Second, I would like to thank Professor Brian Bingham for the introduction to ROS. Professor Bingham, taking your robotics class in the fall quarter gave me a basic understanding of how to utilize ROS and the MATLAB callback functions that helped so much in completing this thesis. Thank you for all of the work and support you give to the students working with ROS at this school.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Robotic vision has recently drawn the attention of many researchers as mobile robotics technology and platform availability have expanded. Robotic vision is the method of equipping a robot with sensors, such as a camera, to obtain images of the environment that can be analyzed to gain visual understanding of the environment. With previous levels of technology, conventional visual processing methods used two-dimensional images, such as a camera image, to analyze a three-dimensional environment. This did not provide enough information about the environment; however, with increasing availability of new technology, objects and humans can be observed in three dimensions. Observing in three dimensions has helped eliminate errors from estimating a three-dimensional environment with a two-dimensional picture of that space.

More broadly, human detection and tracking have been extensively studied in the realm of computer vision and have many applications to mobile robotics and robotic vision. Computer vision seeks to use sensor data to automate the tasks of the human visual system. It seeks to use information from a single image or a sequence of images to analyze and understand the environment depicted. In human detection and tracking specifically, the objective of computer vision is to find a human and follow the human's movements using a sequence of images. Both cameras and range-sensors are popular sensors for this task; however, red, green, blue camera and depth, RGB-D, sensors that provide both depth and image data, are becoming more popular for this task due to increasing availability and affordability.

There are widespread challenges with human detection in images or videos. Variations in posture, light conditions, occlusion, and the complexity and cluttered state of the background all have significant effects on the ability to detect a human from sensor data. Having a large sensor suite of multiple types and streams of data can help alleviate some of the challenges; however, multiple sensors can be costly. Additional challenges are seen in human detection when incorporated with robotics due to the added motion of the robot and the change in background environment. Effects from this motion can be seen on the images or videos collected.

The most widespread and successful RGB-D sensor is that developed by Microsoft for the Microsoft Kinect. The Microsoft Kinect sensor was originally released as an accessory for the Xbox 360 to allow for interactive game play, allowing humans to interact with games using gestures and body motion. This RGB-D sensor provides both a camera image as well as a corresponding depth image. The sensor developed by Microsoft was released to the public along with its software development kit. The Microsoft Kinect's software, in its ability to do skeleton tracking pose estimation, was state of the art. Due to its low cost, availability, and software capabilities, the Kinect has become an influential and widely used sensor for developers and researchers in robotics being used in the fields of simultaneous localization and mapping, autonomous navigation, and human tracking and following.

The Robot Operating System (ROS) is an open-source operating framework, and its use has become widespread both in academia and the public and private sectors. It provides tools and libraries for researchers developing software to run robots and their sensors with packages available for many commercially available robotic systems. With a large collection of drivers and fundamental robotics algorithms, as well as tools to visualize robot state, sensor data, and debug faulty behaviors, the framework gives a starting point for research and development. The open source operating system allows and encourages the sharing and collaboration by multiple users in developing software allowing users to piece together individual small packages that work for their individual systems and incorporate their own algorithms, saving the user time.

A. MOTIVATION

Person detection is a fundamental task for many robots, intelligent vehicles, and interactive systems that share their environment space with humans. Service robots must detect, identify, and track humans in a complex environment. These robots should have capabilities allowing them to respond to hand gestures and specific human actions. The ability to detect and track the human, specifically the human skeleton, is a key component of recognizing gestures and actions. It is also a major part of the human following task that these robots are designed to complete. If the robot can detect and track human motion, the robot will not need to build an environment map or the user to have to input a target location.

In the military realm, robots are used to conduct dull and dirty tasks and decrease risk to the soldier, but most of these robots must be manually driven. A human following robot that did not have to be manually driven would decrease a soldier's workload. With the necessary sensor suite on the mobile robot, the robot would also have the ability to run autonomously locally without needing to send and receive data from the command location. Autonomous robots that can assist medical personnel and evacuate personnel on a battlefield are other areas that will benefit from human detection and tracking.

The Legged Squad Support System (LS3) developed by Boston Dynamics is one example of a robot that was designed to work with the military [1]. The LS3 pack mule style robot was designed to be able to automatically follow humans and respond to simple voice commands. Avoiding a joystick and computer screen control was important because this allows the soldier to focus on the mission at hand; however, development on the system was shelved in 2015 due to noise level problems, among other issues [2].

Human detection and tracking also has security applications. The ability to detect and track a human could be integrated into patrol robots, increasing a facility's perimeter security posture while simultaneously decreasing manning requirements.

B. PREVIOUS WORK

Since the release of the Microsoft Kinect sensor and its software development kit, researchers have continuously been working to implement human detection, tracking, and following algorithms using sensor data. Some researchers simply use the depth and camera raw images available from the Kinect sensor to conduct human detection and following using computer vision techniques. Others use the skeleton tracking capabilities of the Kinect which outputs position and orientation of the person's joints and either look to improve on this algorithm or combine this ability with auxiliary methods to create a more robust tracking system [3].

Many approaches use the Kinect sensor for human detection and tracking utilizing the raw RGB-D data. Some methods use a model-based approach. First, regions where a person may exist are determined from the depth data. Inside that region specific body parts like the head and torso are attempted to be matched with a model [4]. Several researchers have looked into using histograms of depth (HOD) data as well as histograms of oriented gradients (HOG) in RGB data [5], [6]. Others do human segmentation using background subtraction methods [7], [8]. These methods cover the spectrum of using only the depth data, only the RGB data, or using both streams of data together.

Use of the Kinect sensor's skeletal tracking for mobile robotic completion of the human following task also takes multiple forms. Babaians et al. approach was to combine the skeletal tracking with an auxiliary vision tracker, OpenTLD, to create a more robust system that could function even when the Microsoft Kinect skeleton tracking failed [9]. A similar method using Camshift as the auxiliary tracker was capable of correcting the tracking when the skeleton tracking failed [10]. The authors found that their tracking system had greater success than the skeleton tracking system alone with a cluttered environment or when the user moved with their back facing the robot. Other researchers focus on implementing a Kalman filter to estimate the position of the human and decrease noise from the skeletal data collected [10], [11].

At the Naval Postgraduate School, previous work has been conducted with the Microsoft Kinect RGB-D sensor in the Electrical and Computer Engineering Department focusing on the ability to use the Microsoft Kinect sensor for obstacle avoidance. The Kinect sensor was capable of detecting thin or narrow obstacles that the onboard sonar sensor of the mobile robot could not detect [12]. Research into the ability to interface the Microsoft Kinect with ROS for mobile robot autonomous navigation and map-building was also conducted [13]. It was shown that it was feasible to use ROS to conduct Simultaneous Localization and Mapping (SLAM) and autonomous navigation without a GPS or simulated indoor GPS; however, neither of these two theses focused on the human tracking capabilities of the Microsoft Kinect sensor.

C. PURPOSE AND ORGANIZATION OF THESIS

The purpose of this thesis is to investigate the feasibility of implementing human following with a Kinect RGB-D sensor on an indoor mobile robot using a specific development environment involving ROS and MATLAB. The thesis is divided into six chapters. An overview of ROS and the MATLAB toolbox used in the project as well as the hardware, the Microsoft Kinect, the Pioneer P3-DX, and the Computer Processing Units is given in Chapter II. The system development and integration of the ROS system and packages is discussed in Chapter III. The approaches taken for the two human tracking methods implemented as well as the controller for the robot are explained in Chapter IV. The results of the tracking methods are focused on in Chapter V. Lastly, the work of the thesis is summarized and future work in the areas of this thesis at NPS are presented in Chapter VI.

THIS PAGE INTENTIONALLY LEFT BLANK

II. DESCRIPTION OF SOFTWARE AND HARDWARE SYSTEMS

The software and hardware design put into operation for this thesis is described within this chapter. Firstly, the software used is explained with descriptions of the Robot Operating System and the MATLAB Robotics toolbox. Secondly, the hardware used is detailed with descriptions of the P3-DX mobile robot, the Microsoft Kinect sensor, and the Computer Processing Units.

A. SOFTWARE

1. Robot Operating System

To gain an understanding of the Robot Operating System, it is important to understand its history, philosophy, and parts. Quigley, Gerkey, and Smart, authors of *Programming Robots with ROS*, give a brief history and introduction to the operating system [14]. Quigley and Gerkey are cofounders of the Open Source Robotics Foundation which maintains and develops ROS. In the mid-2000s at Stanford University, various projects involved the creation of prototypes of a "flexible, dynamic software system" similar to what ROS has become [14]. The projects involved integrative embodied Artificial Intelligence (AI), and some examples include the Stanford AI Robot and the Personal Robots program. The robotics research community saw the need for an open collaboration framework. In 2007, Willow Garage, Inc., a robotics incubator neighboring Stanford, provided resources to extend the earlier concepts investigated in these Stanford projects; however, countless researchers also contributed to the making of the core of ROS and its fundamental software. Multiple institutions, on multiple robotic platforms, developed the ROS framework concurrently.

Understanding the philosophical aspects of ROS are key to comprehending how and why the system is widely used. All aspects of ROS's development philosophy support its primary goal of encouraging sharing and collaboration. Quigley, Gerkey, and Smart in [14] give five aspects which are explained in detail: the use of peer-to-peer connections, the tools-based system, the multilingual approach to programming languages, the objective of being thin, and the free and open source licensing.

Peer-to-peer connections allow for easy scalability as the amount of data in a system increases. ROS functions using “small computer programs that connect to one and another and continuously exchange messages” [14]. With no central routing system, the messages travel directly from one program to another. Although a roscore service provides information to each node, the messages are not sent through the roscore. The ROS master established with roscore is only used to tell the node where other nodes and data streams are located to allow for the pee-to-peer connection.

The focus on a tools-based system enables large numbers of users to use the same programs. Individual tools are small and generic [14]. A separate program is necessary for logging data, visualizing system interconnections, plotting data streams, etc. Quigley et al. explain how the large set of tools available in ROS allow for easy visualization of robot states and algorithms, debugging of behaviors, and recording of sensor data, encouraging developers to create new and improved implementations of each desired task. In newer versions of ROS, some tools have been combined into a single process for efficiency, creating a cleaner interface for the developer.

The multilingual approach to programming languages which ROS takes allows individual researchers to choose the language in which they work. Continuously, researchers dispute which programming language is best suited to complete a task. As such, the developers of ROS believed that situational requirements determine the necessity of using different programming languages [14]. ROS’s capabilities include the ability for the developer to write software modules in any scripting language that is supported through a client library. The two most largely used and documented client libraries are created for Python and C++; however, client libraries exist for more than ten programming languages [15].

The thin objective of ROS allows the operating system to be easily integrated with other frameworks. Being thin implies that the setup of ROS encourages users to create and use libraries that are standalone and then to simply wrap the libraries, allowing them to send and receive from other ROS modules [14]. The purpose of this thin objective is to allow the reuse of programs outside of ROS with other robot software frameworks.

The final and paramount aspect of ROS's philosophy is its being free and open source. Quigley et al. explain that the core of ROS is released under a BSD license. The license allows for both commercial and noncommercial use. Any individual or group can start a ROS repository on their own server, and it is up to the individual or group if they wish to make the repository available to the public. Developers can share their personal adaptations and codes using GitHub, a repository hosting service. Furthermore, ROS includes a wiki site including documentation, tutorials, and a discussion forum to which users can contribute and submit questions.

The specifics of the installation of ROS and the packages utilized are further discussed in Chapter III.

2. MATLAB

Worldwide, MATLAB is widely used by engineers and scientists. In education specifically, MATLAB is used widely as a basic programming language for mathematical computation. MATLAB is seen as a user-friendly environment with its method of processing, evaluating, and graphically displaying numerical data. With its "matrix-based language" it is a "natural way to express computational mathematics" [16]. With a collection of toolboxes available, both with the general download as well as others for purchase, MATLAB has tools varying from control systems to signals processing and communications. Thorough documentation available both in command line tools like the `help` command as well as on the MathWorks Documentation site makes using MATLAB's many toolboxes easier [17].

Prior to 2015, multiple groups produced solutions to the problem of integrating ROS with MATLAB; however, none of the libraries, or bridges as they were termed, caught on [18]. The Robotics System Toolbox, released in MATLAB R2015a, filled the void and became the leading method of interfacing MATLAB with ROS.

The toolbox, released in 2015, includes a wide variety of tools providing for development of autonomous mobile robotic applications with both algorithms and an interface between ROS and MATLAB. The toolbox includes algorithms for path planning and following as well as map representations in the form of Binary Occupancy

Grids [19]. The interface between ROS and MATLAB allows for access to ROS functionality within MATLAB. When MATLAB communicates with ROS, it acts as one node able to communicate with other nodes by registering with the ROS master. In the initialization phase, the node attempts to connect to the ROS master at the local host, and if one has not already been opened, it starts a new core in MATLAB. Once the node has been established, bidirectional communication in real time through the use of publishers and subscribers allows for the exchange of many supported message types across the network. The contents of a ROS message can be viewed and edited using functions included in the toolbox. Lastly, the interface allows for the reading of rosbag data files that store ROS message data. This allows for collection of data during testing and later analysis, visualization, and processing of data post testing in an easy user friendly format.

B. HARDWARE

1. P3-DX Mobile Robot

Mobile robots are widely available for educational purposes and research. Starting in 1995, Adept MobileRobots began commercially selling mobile robots beginning with the Pioneer 1. In this thesis research we use the Pioneer 3 DX, P3-DX, mobile robot which, without added hardware, is shown in Figure 1. The P3-DX is a compact differential-drive mobile robot with two wheels that is designed for use in indoor laboratory or classroom settings [20]. The robot comes with a complete software development kit, Pioneer SDK. The Advanced Robot Interface for Applications (ARIA) is used as the main library tool for interfacing with the mobile robot [21]. This core library "provides an interface and framework for controlling and receiving data from all MobileRobots (ActivMedia) robot platforms" [21]. Notably, it includes an open source framework that allows for client-server network programming.

The P3-DX can travel at a maximum forward or backward speed of 1.2 m/s [22]. The robot is able to turn in location with a 0.0-cm turn radius. The operational payload of the robot, the allowable added weight, is 17 kg. The robot runs on up to three batteries at a time with a battery voltage of 12.0-V. The robot communicates to the connected libraries through a serial connection.

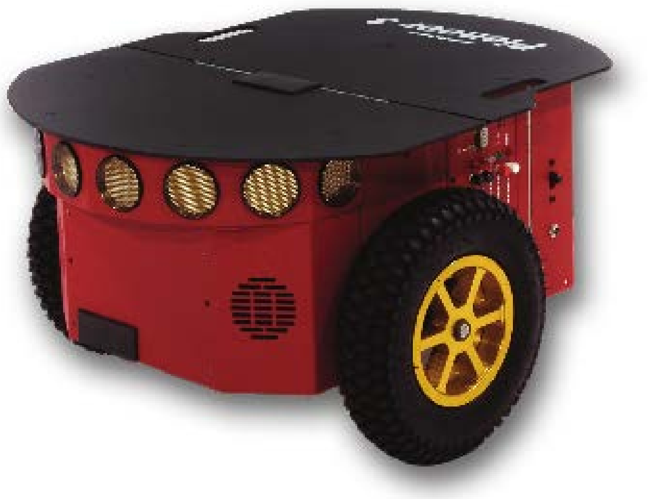

Figure 1. P3-DX Mobile Robot as Distributed by Manufacturer. Source: [23].

2. Microsoft Kinect

Microsoft Kinect was released in 2010 as an input device to the Xbox game console. It was revolutionary in its ability to allow the user to interact with games without any controller through its human detection algorithm. Before February 2012 and the release of the Kinect Software Development Kit (SDK) for Windows, the Kinect was already being used for research and non-gaming applications. Researchers realized that the three-dimensional (3-D) depth image available from the Kinect was comparable, at a much lower cost, to much more expensive 3-D depth methods such as stereo cameras or time-of-flight cameras [3]. The "complementary nature" of the RGB data with the depth data also allowed for interesting new research. Web based discussion communities like KinectHacks.net arose and saw wide use [24]. Large numbers of projects and applications for the Kinect were posted. Papers were also published before February 2012 that used the Kinect as the main sensor.

With the release of the SDK for Windows, it became easier for researchers and hobbyists to work with the Kinect sensor data directly. The release gave access to the raw camera data streams as well as the skeletal data from the 3-D human motion algorithm; however, other libraries for working with the Kinect exist. Two other available Kinect tools available are OpenNI and OpenKinect [3]. Unlike the Microsoft SDK, which is only available on Windows, the OpenNI tool works together with a middleware called NITE and works on multiple platforms, an attribute that is important for this thesis. The OpenNI tool and NITE middleware can be run on a Linux computer. A key difference between the two libraries is that the Microsoft SDK does not require calibration before it begins human tracking, whereas the OpenNI library requires the user to stand in a specific calibration pose to initialize the tracking [3]. A comparison of the two libraries is shown in Table 1.

Table 1. Comparison of the OpenNI Library and the Microsoft SDK. Adapted from [3].

	OpenNI	Microsoft SDK
Camera calibration	Yes	Yes
Automatic body calibration	No	Yes
Standing skeleton	Yes (15 joints)	Yes (20 joints)
Seated skeleton	No	Yes
Body gesture recognition	Yes	Yes
Hand gesture analysis	Yes	Yes
Facial tracking	Yes	Yes
Scene analyzer	Yes	Yes
3-D scanning	Yes	Yes
Motor control	Yes	Yes

The NITE library was developed by the company PrimeSense. They developed the Prime Sensor Development Kit, similar to the Microsoft SDK. NITE Algorithms user guide [25], included within the zipped NITE file folder, contains a basic description of the capabilities of the package. Specifically in the discussion of user segmentation and skeleton tracking, the user guide discusses known issues. One known issue that can produce inaccuracies in user segmentation is if the sensor is being moved while the user segmentation is active. Although automatic calibration was added in NITE version 1.5, versions before it did not have the ability of autocorrelation, and a psi pose had to be used [25]. The NITE joint locations used for skeleton tracking are shown in Figure 2.

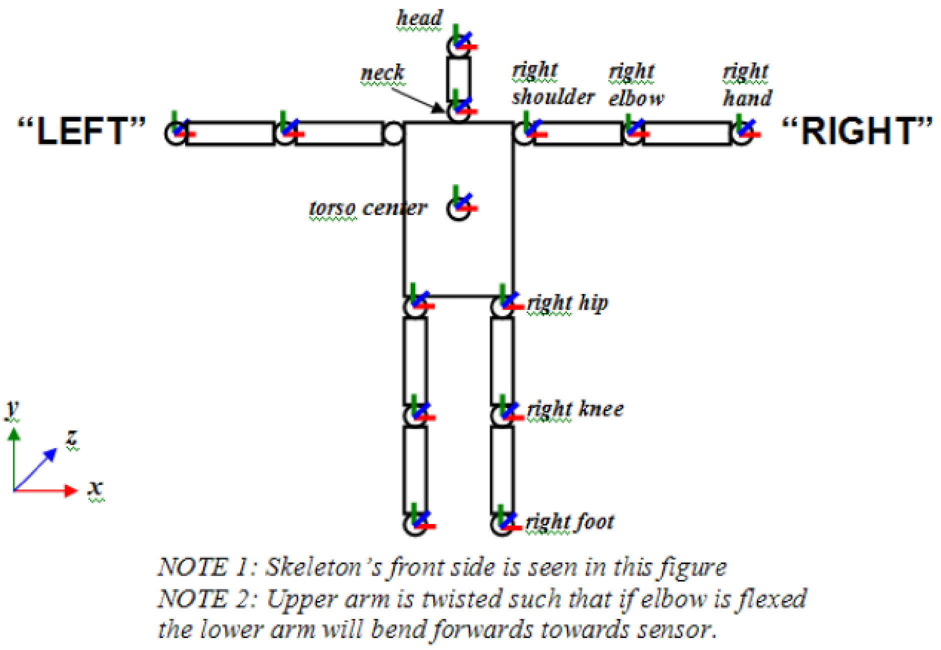
head
neck
right shoulder
right elbow
right hand
"LEFT"
"RIGHT"
torso center
right hip
right knee
right foot
y
z
x
NOTE 1: Skeleton's front side is seen in this figure
NOTE 2: Upper arm is twisted such that if elbow is flexed the lower arm will bend forwards towards sensor.

Figure 2. NITE Algorithm Joint Definition. Source: [25].

The Microsoft Kinect sensor for Xbox 360 contains advanced sensing hardware for its relatively low cost. The Kinect sensor consists of an infrared projector, infrared camera, and a color camera [3]. The location of the hardware aboard the sensor can be seen in Figure 3. The depth sensor functions by the infrared projector sending an infrared (IR) speckle dot pattern into the 3-D scene [3]. As explained in [3], the infrared camera captures the reflected infrared speckles, and a depth map returning the distance of an object relative to the sensor is determined using the return from the infrared camera as well as a calibration file. This depth data is a 640×480 pixel map published at a rate of 30 frames/s. The authors express the angular field-of-view of the sensor as 57° horizontally and 43° vertically. Similarly, the color camera produces an RGB image with three color components, the size is 640×480 and is also operated at 30 frames/s [3].

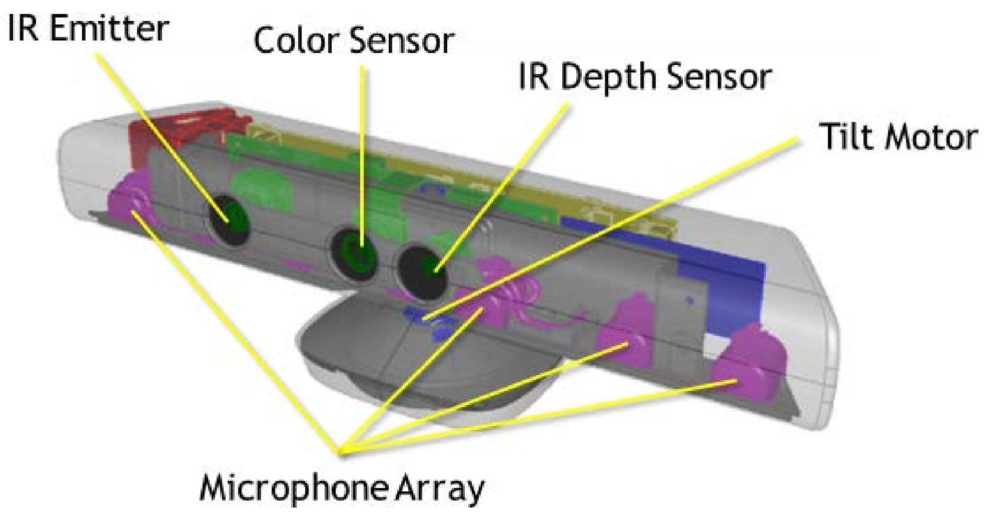

Figure 3. Microsoft Kinect Hardware Configuration. Source: [26].

In 2012, the Microsoft Kinect and the Windows SDK library were analyzed for performance by Livingston et al. [27]. For their analysis, the Kinect sensor was mounted on a flat surface for testing and the sensor was not in motion. The optimal range as suggested by Microsoft is 1.2 m to 3.5 m, but they found in their tests that a skeleton could be acquired in the range of 0.8 m to 4.0 m [27]. The investigation determined that for any skeleton past the 4.0 m mark, the location was set to zero. The Windows SDK limited depth tracking of the skeleton to the range of 4.0 m.

3. Computer Processing Units

In this thesis research, two computers and a local Wi-Fi network were utilized. The first computer is a desktop computer reimaged to run Ubuntu Linux 14.04 (Trusty Tahr). This was used to run the roscore master for the ROS system, MATLAB, and other ROS packages that did not have to be run on the computer connected to the robot and sensor. With the use of secure shell protocols (SSH), the second computer can be run from the desktop. The code used for the thesis research was run on the second computer that was initialized inside a shell on the desktop computer.

The second computer is a SUMICOM Small Computer, design S675, reimaged to run Ubuntu Linux 14.04 (Trusty Tahr). The computer has four USB 3.0 ports, two USB

2.0 ports, and a CD Driver [28]. Connection to the Kinect sensor is made through the USB 3.0 port. The computer also has one RS232 (COM) port for serial communications. The mobile robot is connected to the computer through the RS232 (COM) port. The minicomputer has a total memory size of 16 GB and uses a 12.0-V power supply provided by the robot. To connect to the wireless network, the computer is equipped with a built-in Intel LAN controller. The small computer was physically mounted atop the mobile robot. ROS packages to run the skeleton tracking, Kinect sensor, and the mobile robot were run on this onboard computer.

A local wireless network was established using NETGEAR wireless LAN products. A NETGEAR Web Safe Router was used to allow one broadband connection which allowed the two computers to communicate. A 5.0-V power supply was provided to the router. The router was connected to the internet through an Ethernet cord plugged into both the NPS wired network and into the internet connection on the back of the router. Four LAN connections were available on the router, but only two of the LAN connections were used. One LAN connection was made directly to the desktop computer through an Ethernet cable. The other LAN connection was made to the Wireless N150 Access Point. The location of the connections on the router can be seen in Figure 4. A Wireless N150 Access Point provided connectivity to the network within a provided range. It acted as a bridge between the wired LAN system and the wireless client, the SUMICOM small computer.

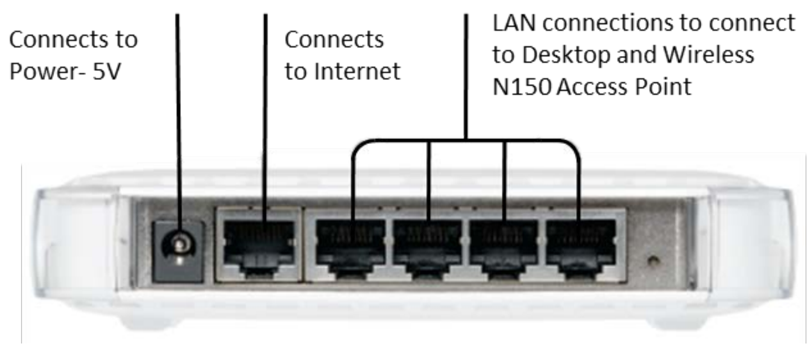
Connects to Power- 5V
Connects to Internet
LAN connections to connect to Desktop and Wireless N150 Access Point

Figure 4. NETGEAR Router Connections. Adapted from [29].

C. SUMMARY

An explanation of ROS and MATLAB were given within this chapter. The hardware used within the thesis was also described within this chapter. The relation between the different hardware pieces is visualized through a system diagram in Figure 5. Each hardware piece is shown with its method of connection depicted.

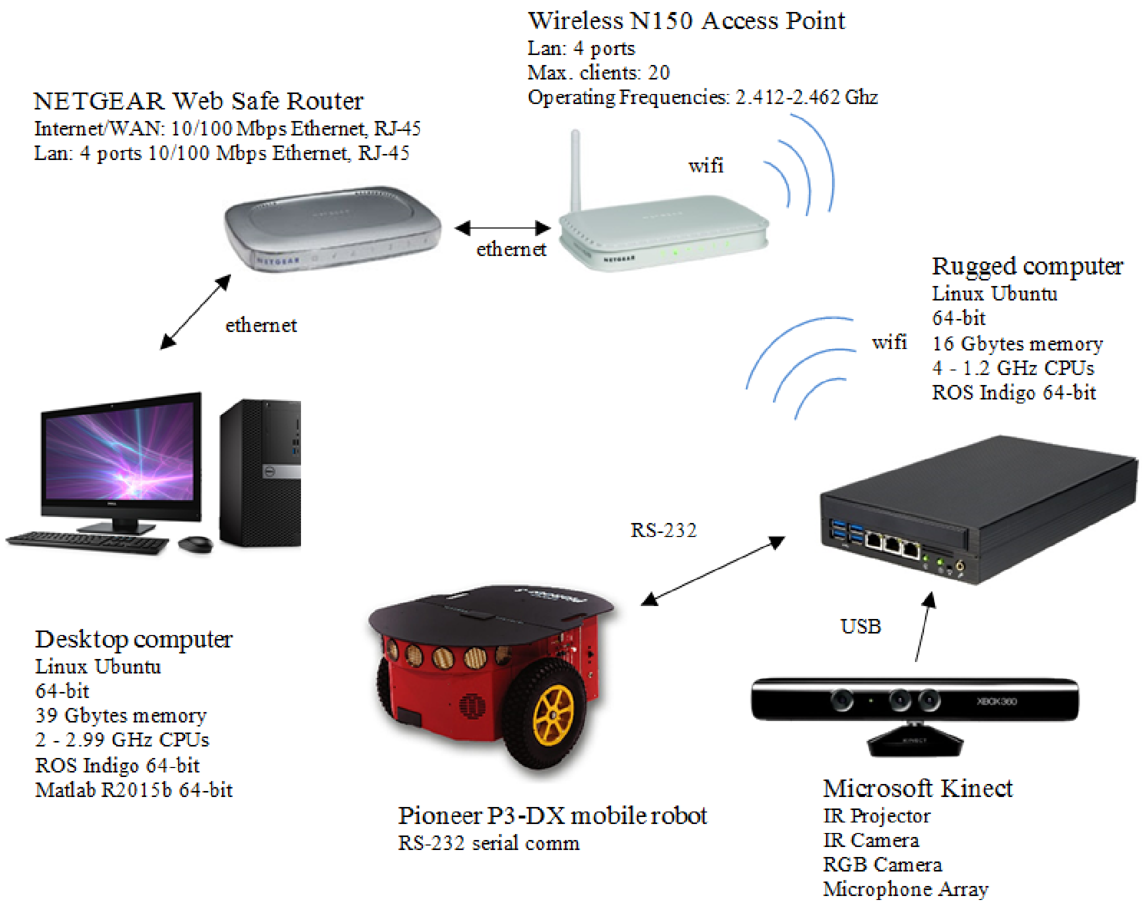

Figure 5. System Diagram. Adapted from [23], [28], [30]–[33].

III. DESCRIPTION OF SYSTEM DEVELOPMENT AND INTEGRATION

The development of the software systems is discussed within this chapter beginning with the Ubuntu Linux operating system and the ROS installation. The setup of the ROS network over the two computers is explained. Then, the download and setup process of the ROS packages and repositories is discussed.

A. BASE INSTALLATION OF UBUNTU AND ROS

To set up the desired software systems, the computers were reimaged to run Ubuntu Linux 14.04 (Trusty Tahr). The Ubuntu 14.04 software was downloaded from the Ubuntu website as an ISO image file. Using the help community pages, we followed tutorials which explained how to burn the ISO image file to the DVD [34]. With this imaged DVD located in the disc drive of the computer, the computer's boot menu was accessed [35]. The boot order for the computer was changed, putting the disc drive first. After we restarted the computer, the opening screen appears and gives the option of trying Ubuntu or proceeding directly to installation. At this point the Ubuntu Linux operating system was installed and the Microsoft Windows was removed from the computer. The installation process was repeated for both the desktop computer and the SUMICOM minicomputer.

Before beginning installation of any of the ROS packages, configuration changes had to be made to the base Ubuntu installation. The Ubuntu repositories store programs available to Ubuntu in software archives. Opening the Ubuntu Software Tab allows the user to check under the software tab the restricted, universe, and multiverse repositories to allow their access [36].

Due to ROS's open source and large user wiki, an installation page for Ubuntu platforms exists for each ROS distribution. The chosen ROS distribution, ROS Indigo Igloo, was released on July 22, 2014. The distribution primarily targeted the Ubuntu 14.04 release. The "Ubuntu install of ROS Indigo" was followed to ensure that all lines were written properly in the command line. The installation first checks that the Debian package is up-to-date using the update command. Next, the full desktop installation of ROS Indigo was installed. The full desktop installation, which is recommended, gives access to “ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators and 2D/3D perception” [37]. Next, rosdep was initialized and updated. Rosdep enables easy installation of system dependencies for compiled sources. For some core components of ROS to function, rosdep must be installed. The environment is set up by adding a source line to the .bashrc file. The .bashrc file runs every time a new shell is launched. By adding the source line of code, the ROS environment is set up automatically each time a new shell is opened. Lastly, the rosinstall command line tool was downloaded allowing easy download of some source trees for ROS packages with a single command [37].

The SSH client and server also had to be installed. SSH is used to allow the user to remotely log in to computer systems securely over an unsecured network. Specifically, the SSH protocols allow for the user to remotely log in to the computer aboard the robot from a terminal window open on the desktop computer. This allows for the mobile robot to operate without an onboard monitor. SSH was installed to the computer using terminal install commands.

With the two computers on a single network created by the NETGEAR router, the computers can run a single ROS master, roscore. Tutorials exist explaining how to run the master for a system on one computer. The “Running ROS across multiple machines” tutorial helps to configure multiple machines to use a single master [38]. The Internet Protocol (IP) addresses of the desktop computer and SUMICOM mini-computer were set using the NETGEAR router to 192.168.0.2 and 192.168.0.3, respectively. Using the ROS Network Setup documentation, we completed name resolution by exporting the ROS_IP and ROS_HOSTNAME as well as configuring the etc/hosts file so that the machines could find each other [39]. The file lines were added to the .bashrc and etc/hosts file for the desktop and the minicomputer aboard the robot. The lines in Figure 6 were added to the .bashrc file and the etc/hosts file for the desktop computer, and the lines in Figure 7 were added to the two files on the minicomputer. With the addition of the lines, the computers can locate each other on the network. With the edits to the two files, the minicomputer can be run from the desktop computer using SSH, and a single roscore master can be run for the two machines.

```
# lines added to the  /.bashrc file:

source /opt/ros/indigo/setup.bash
source ~/catkin_ws/devel/setup.bash
export ROS_MASTER_URI=http://192.168.0.2:11311
export ROS_IP=192.168.0.2
export ROS_HOSTNAME=192.168.0.2

# file lines added to the /etc/hosts file:

192.168.0.2 basecomp
192.168.0.3 ragreenb@cslab cslab
```

Figure 6. Lines Added to Files on the Desktop Computer

```
# lines added to the /.bashrc file:

source /opt/ros/indigo/setup.bash
source ~/catkin_ws/devel/setup.bash
export ROS_MASTER_URI=http://192.168.0.2:11311
export ROS_IP=192.168.0.3
export ROS_HOSTNAME=192.168.0.3

# lines added to the  /etc/hosts file:

192.168.0.3 cslab
192.168.0.2 ragreenb@basecomp basecomp
```

Figure 7. Lines Added to Files on the Computer onboard the Mobile Robot

A ROS workspace must be created to allow the download and installation of ROS packages. Catkin workspaces in ROS Indigo allow catkin packages to be built. The “Creating a workspace for catkin” tutorial was used to set up the environment [40]. Once the workspace has been created and built, ROS packages can be cloned from a git repository or installed using the command line tool and built in the workspace. With a single catkin workspace, the setup file may be added to the .bashrc file to run each time a shell is opened. The second line shown in Figures 6 and 7 was added to the .bashrc file, which sources the setup file for the workspace.

Lastly, git must be installed. GitHub is used widely by the ROS community. With a free account, a user can create public repositories. Many open-source ROS packages are stored and are available as git repositories. Installing git using the command line tool gives the ability to clone these repositories. Cloning a git repository creates a copy within the folder in which you are currently located. Git repositories are cloned into the src folder of the catkin workspace.

B. ROSARIA

The RosAria node authored by Srećko Jurić-Kavelj is available for ROS distribution Indigo. The node provides an interface with ROS for Adept MobileRobots which use the open source ARIA library [41]. First, the ARIA library must be installed from Adept MobileRobots. The ARIA software ARIA 2.9.1 for Ubuntu 12.04.2, or newer, for a 64- bit architecture was downloaded. The RosAria node was cloned from source using git. By using rosdep, we also installed the necessary dependencies. Lastly, the node was built using catkin.

To test and run the RosAria node, the “How to use ROSARIA” tutorial was referenced [42]. With the mobile robot connected to the computer using a RS232 serial communications link, the rosrun command is issued with the port parameter specified defining a serial connection over COM1. The RosAria node publishes on multiple topics including pose and battery voltage. The /pose topic is of the ROS message type nav_msgs/Odometry. The message contains the position in xyz-coordinates as well as the rotation or twist of the robot. Both the pose and twist messages also contain covariance values. The pose of the robot is set to [0, 0, 0] at the start of the RosAria node, setting the [0, 0, 0] world location at the robot’s starting location. The battery voltage is a float message of type std_msgs/Float32 that gives a measurement of the battery voltage in DC. The node subscribes to the topic /cmd_vel, the topic on which new velocity commands to the robot can be published. When a command is sent over the topic, the desired velocity is set in ARIA. The robot obtains and maintains the velocity sent for up to 600 ms unless another velocity command is received. As such, the velocity commands to the robot are only necessary for changing its speed [41].

C. P2OS AND AMR-ROS-CONFIG

Two other ROS packages must be installed to ensure the ability to run the Adept Mobile Robots correctly. The first package, p2os, is installed from source with git. The p2os package from allenh1 contains rviz robot models for simulation [43]. The package was not updated to run with ROS Indigo; however, the models contained within the package are up to date. The second package cloned to the src subdirectory of the catkin workspace is the amr-ros-config package. The repository contains Unified Robot Description Format (URDF) and launch files as well as other ROS configurations for Adept Mobile Robots [44]. The URDF files in the description subdirectory of the amr-ros-config are based on sources, including the p2os package first installed. The amr-ros-config package also contains a folder with examples for running gazebo, the simulator used with ROS.

The URDF files are ROS's method for storing robot model descriptions. The P3-DX robot's URDF model is available describing the locations and orientations of the different robots transforms. Having a local copy of the URDF for the mobile robot was important because the robot's sensor had to be added to the URDF. Lines added to the URDF allow for the programs to understand where the sensor was located and oriented compared to the robot. The additional lines added to the URDF of the robot to include the Kinect sensor were adapted from the previous thesis work of Capt. Lum of the United States Marine Corps [13]. Using ROS tools like xacro, check_urdf, and view_frames, we built the URDF file and checked for a successful creation [45]. The successful transform tree built by the URDF file once the Kinect sensor was added is shown in Figure 8.

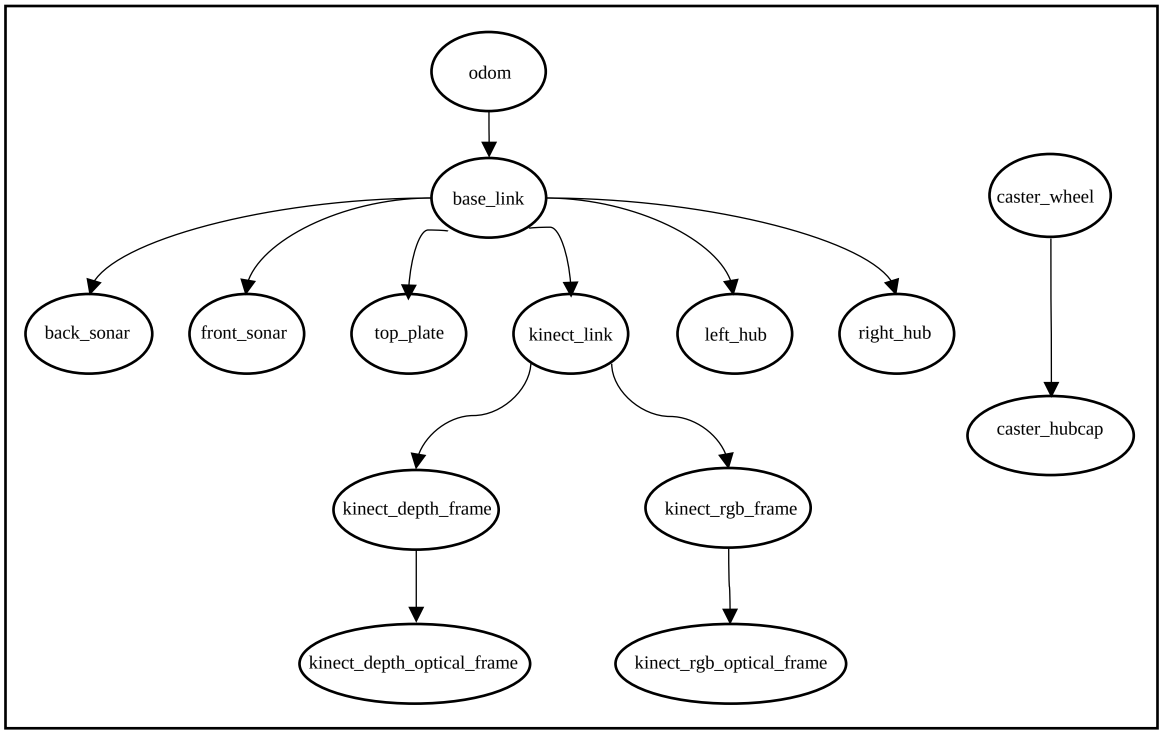

Figure 8. Visualization of the Full Tree of Coordinate Transforms for the P3-DX Mobile Robot with Kinect Sensor

D. OPENNI STACK AND RGBD_LAUNCH

With the packages downloaded to be able to run and visualize the Pioneer robot, the next step was to download the packages necessary to run the Microsoft Kinect sensor. The packages were downloaded to both the desktop computer and to the computer onboard the mobile robot. For initial testing of the sensor, the Microsoft Kinect was plugged into the desktop, but during actual testing and robot motion, the Microsoft Kinect was plugged into the computer onboard the P3-DX. As discussed in Chapter II, the OpenNI library works with a computer running Linux and is well documented in its use with ROS. Starting with the ROS Hydro distribution, the distribution before ROS Indigo, we know much of the functionality of the original openni_stack was moved to the rgbd_launch package [46]. This was to allow other drivers to use the same code. As such, the openni_launch package contains a single launch file which starts RGB-D processing through other nodes. The openni_launch file can be used with any OpenNI-compliant device. For the launch file to run successfully, the openni_camera ROS driver must also be installed. The node created by openni_camera and opened with openni_launch simply publishes the raw data from the sensor [47]. The nodelets that are established with the launch file convert the raw depth, RGB, and IR data streams from openni_camera to depth images, disparity images, and point-cloud data using the rgbd_launch nodes.

In order to run the Microsoft Kinect, a driver must also be installed from source. The PrimeSensor Modules for OpenNI are located within a git repository [48]. The git repository is cloned to the computer, and the necessary files must be extracted and installed. The lines of code in Figure 9 were used to clone and install the necessary driver for the Microsoft Kinect sensor.

```
# Lines ran in the terminal window to clone, extract, and install PrimeSensor Modules


$ git clone https://github.com/avin2/SensorKinect
$ cd SensorKinect/Bin
$ tar xjf SensorKinect093-Bin-Linux-x64-v5.1.2.1.tar.bz2
$ cd Sensor-Bin-Linux-x64-v5.1.2.1
$ sudo ./install.sh
```

Figure 9. Terminal Window Command Lines to Clone the Git Repository and Install the Linux Driver

With the installation of the described packages, the Kinect sensor was able to be run from the computers. With the help of the “QuickStart” tutorial for the openni_launch package, the sensor data of the Microsoft Kinect was visualized using rviz and the image_view tool in ROS. The pointcloud data from the Kinect was visualized using rviz, and an example is shown in Figure 10. Left image is the direct view from the sensor, while the right image is a side view of the point cloud data. The depth and camera images from the Kinect were visualized using the Camera in rviz, and an example image is shown in Figure 11. The left image shows the raw depth data. The right image shows the corresponding RGB color image.

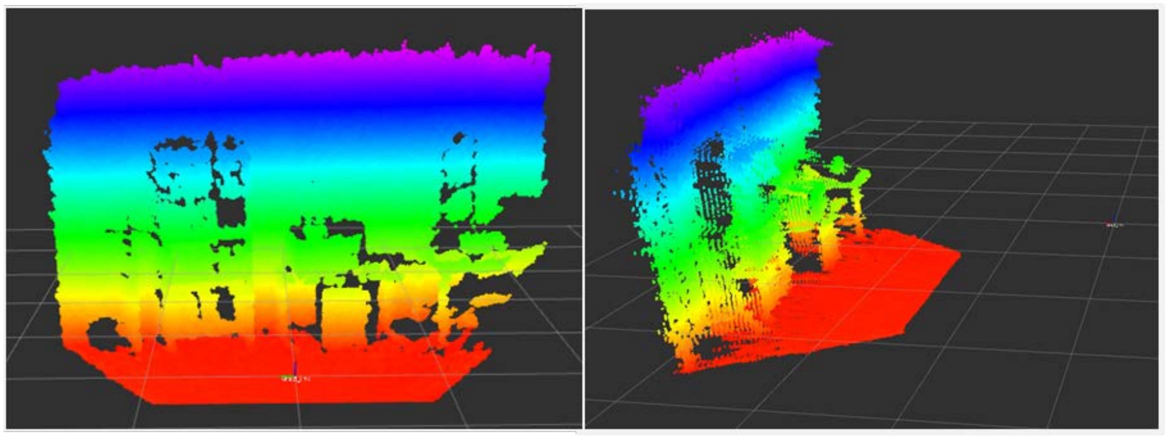

Figure 10. Point Cloud Visualization through rviz

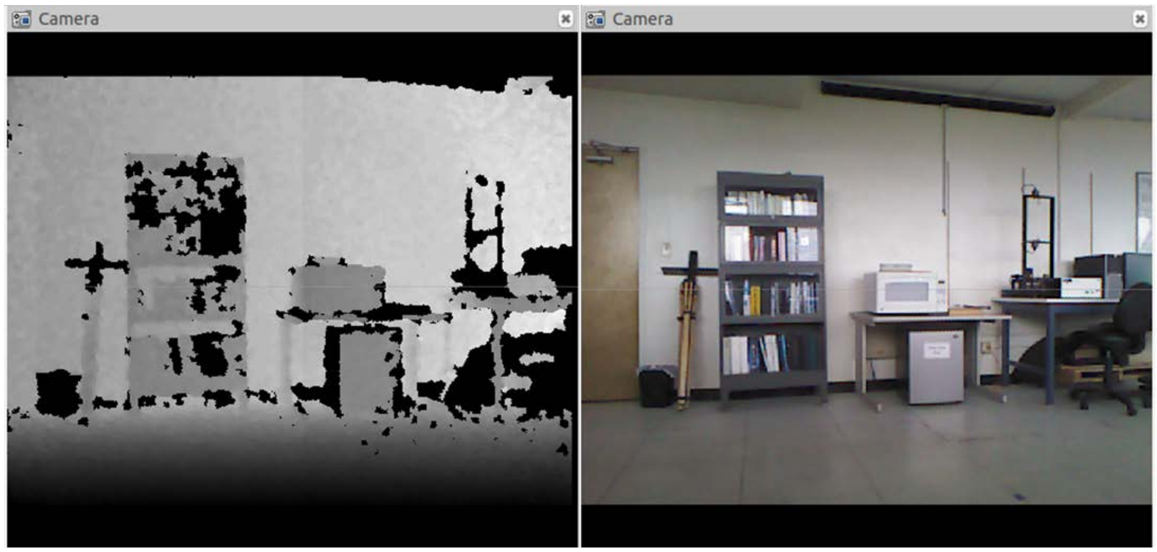

Figure 11. Example Image of the Lab Environment

E. OPENNI_TRACKER

Next, the OpenNI tracker package for ROS was installed. The package openni_tracker, by author Tim Field, is the available node that broadcasts the OpenNI skeleton frames using /tf, ROS coordinate transform messages [49]. Although the package is only updated and maintained through the ROS Hydro release, the package works with ROS Indigo. The Hydro development branch of the package was cloned using git.

In order to be able to run the openni_tracker node, the NITE library software must be manually installed. The NITE software was downloaded from the openni.ru website from the “OPENNI SDK HISTORY” page which contains the files for past OpenNI SDK versions as well as past NITE versions [50]. The openni_tracker package is compatible with NITE v1.5.2.21 and v1.5.2.23; however, only the NITE v1.5.2.23 version for a 64-bit Linux architecture is available on the website. Once the NITE version was downloaded, the files were extracted, and the install file was run. With the NITE library installed, the tracking node package can be successfully run.

The openni_tracker node is independent in that it does not require the openni_launch nodes to be running. The tracking node simply needs the Kinect to have power. Once the node starts and is running, a user must stand in front of the Kinect sensor and hit the psi pose as shown in Figure 12. With calibration complete, the user’s pose is published as a set of 15 transforms [49]. The one parameter that can be set for the node is the camera_frame_id, the name of the frame that all of the transforms will use as a parent.

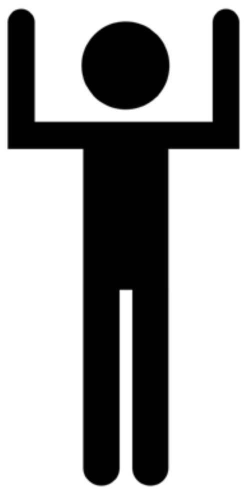

Figure 12. Psi Pose for Startup Calibration of the openni_tracker Node. Source: [51].

F. SLAM AND DEPTHIMAGE_TO_LASERSCAN

For this project, it was desirable to be able to build a map of the environment as the mobile robot travelled through the space. As such, SLAM was a necessary task. The slam_gmapping package for ROS Indigo was installed, which provides a laser-based SLAM node creating a two-dimensional occupancy grid map from laser data and pose data from the mobile robot [52]. The package is a ROS wrapper for OpenSlam's Gmapping. Pose data from the mobile robot exists; however, the Microsoft Kinect does not provide laser data. Nevertheless, another package in ROS, the depthimage_to_laserscan package, takes a raw depth image and outputs a two-dimensional laser scan [53]. The methods for converting the Microsoft Kinect raw data to a laser scan and conducting SLAM using the ROS package were modified from Capt. Lum's prior thesis work [13].

G. SUMMARY

The software development was described within this chapter. The ROS packages and nodes described within this chapter are run together to create one large system. The specific launch files used for each of the nodes can be found in Appendix A. One master launch file was not used due to instability in the launch of the openni_launch node which randomly failed, necessitating a restart of the node. As such, each package was run in a separate terminal window. With the different programs running, the communication can be visualized using a ROS computation graph. The ROS graph built when all of the programs have been launched can be seen in Figure 13. The openni_tracker node is not connected to any of the other topics or nodes because no person was calibrated. Until a person is calibrated, no messages are published by the node. The graph allows the user to visually understand how the different nodes are connected through the topics they publish and subscribe to and allows for troubleshooting in the case of unexpected errors.

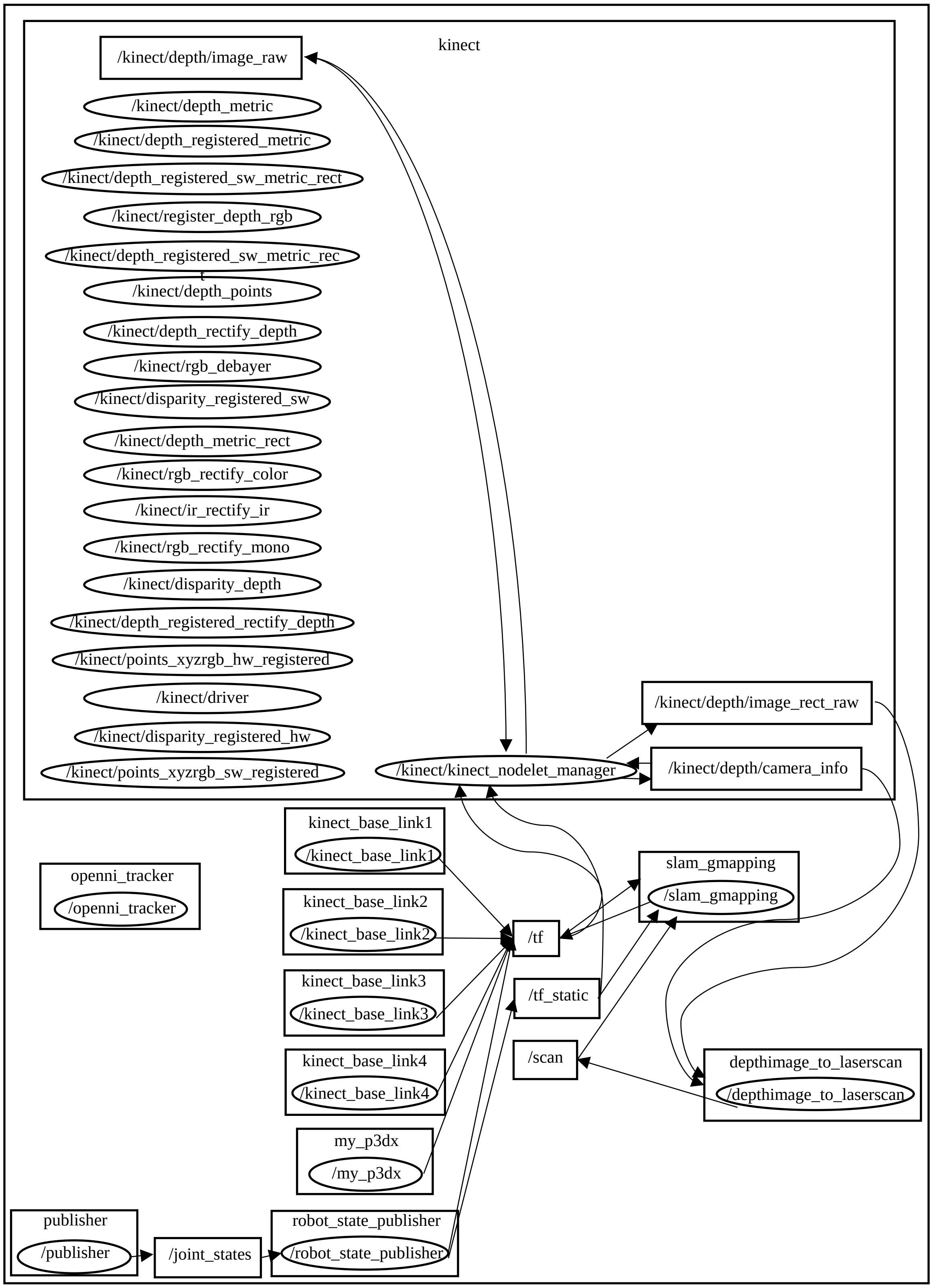

Figure 13. Rosgraph Showing the Nodes and their Connections

THIS PAGE INTENTIONALLY LEFT BLANK

IV. ALGORITHMS

The algorithms used within the thesis are described in this chapter. One MATLAB script file was used to start each experiment, and four separate ROS subscriber callback functions were used inside the script. The first callback function was written to take in the data from the skeleton tracker. The second callback function was made to take as an input the raw Kinect depth data and run image segmentation on that data. The last callback functions were used to drive the robot using the input of its position. The function published a velocity command as an output. The callback functions are discussed in detail.

A. ROS SUBSCRIBER CALLBACK FUNCTIONS AND ROS PUBLISHERS IN MATLAB

With the release of the Robotics System Toolbox, it was necessary to have a method for sending messages to and receiving messages from the ROS network. With a ROS master initialized in MATLAB, a ROS subscriber can be written to subscribe to any topics that already exist in the network, and a ROS publisher can be written to publish messages to the network. The ROS subscriber is able to automatically detect the type of message, unlike the ROS publisher which must have its message type defined. A ROS subscriber can be written to wait to receive a message, or it can be written to execute each time a new message is received as a callback. If using multiple subscribers, the second callback function method must be used to allow other MATLAB code to execute while the subscriber is waiting for a new message. Two options are available when passing information to the subscriber callback functions. The writer can choose to use global variables or arguments can be passed directly to the callback function. Global variables were used to share data due to the large number of arguments passing between the callback functions and the desire to save the data for analysis.

B. KINECT TRACKER CALLBACK

The first callback ROS subscriber used in this thesis research works with the openni_tracker node. The openni_tracker node as written publishes messages on the /tf topic. The /tf topic is also the data stream on which the transforms of the mobile robot are published. As such, the topic on which the openni_tracker node publishes was changed to be /tf_skeleton when the node was ran using ROS command line tools. The command lines used are shown in Figure 14. By publishing on a different topic, we can write a ROS subscriber in MATLAB that only subscribes to the skeleton transforms and not also the mobile robot's transforms.

```
# Lines ran in the terminal window to set parameters for the openni_tracker node and
# start the node

$ rosparam set /openni_tracker/camera_frame_id odom
$ rosrun openni_tracker openni_tracker tf:=tf_skeleton
```

Figure 14. Terminal Command Lines to Set ROS Parameters and Start the Tracker Node

The skeleton tracker transform messages have 15 possible frame names. The frames were shown in Figure 2 in the description of the NITE software. The torso frame was chosen as the position to use for tracking because the frames representing the arms and legs have more variance between measurements. If gesture tracking is desired, a different joint, such as the hand joints, needs to be chosen.

To gain an understanding of the information stored within the transform messages, the ROS subscriber callback was first used to explore the resulting data. The data is stored in a variable named message. Using showdetails, we can explore the contents of the message with an example message shown in Figure 15. The showdetails command not only shows the contents of the message but gives the user an understanding of where the data is located within the object array structure.

```
>> showdetails(message)

  Transforms

    ChildFrameId :  torso_1

    Header

      Seq     :  0
      FrameId :  odom
      Stamp
        Sec  :  1491860524
        Nsec :  129086241

    Transform

      Translation
        X :  2.26913801
        Y :  0.08211563481
        Z :  0.1678627587
      Rotation
        X :  0.4939337634
        Y :  0.5903243741
        Z :  0.4856521535
        W :  0.4143531785
```

Figure 15. Example Message on the /tf_skeleton Message Topic

The transform message data is stored in an object array called Transforms. The structure of the objects stored in the structure is shown in Figure 15. The Transforms object array contains the MessageType, the Header (which contains message about the sequence in which the data came), the ChildFrameId for the message, and the Transform information. By further expanding the structure object as conducted in Figure 16, we see that the Transforms.Transform field contains the Translation of the transform as a ROS Vector3 and the Rotation of the transform as a ROS Quaternion. The ChildFrameId, an object of the Transforms field, is also important to the function because it contains the joint name. The ChildFrameId of each message was compared to a string, allowing the algorithm to ignore any data published with a different frame.

```
# Lines ran in MATLAB command window to explore the ROS message structure
>> message.Transforms

ans =

  ROS TransformStamped message with properties:

     MessageType: 'geometry_msgs/TransformSta…'
          Header: [1x1 Header]
    ChildFrameId: 'torso_1'
       Transform: [1x1 Transform]

>> message.Transforms.Transform.Translation
ans =

  ROS Vector3 message with properties:

    MessageType: 'geometry_msgs/Vector3'
              X: 2.2691
              Y: 0.0821
              Z: 0.1679

>> message.Transforms.Transform.Rotation

ans =

  ROS Quaternion message with properties:

    MessageType: 'geometry_msgs/Quaternion'
              X: 0.4939
              Y: 0.5903
              Z: 0.4857
              W: 0.4144
```

Figure 16. Example Exploration of the Structure Fields of the /tf_skeleton Message Type

The Translation object contained an X, Y, and Z position that was relative to whichever frame the parent frame was set. Initially it was thought that this should be set to the kinect_link frame or one of its children frames; however, the Microsoft Kinect transform frames are initialized at the start location of the mobile robot and do not move with the odom frame as the robot moves. The frames of the mobile robot after the robot has been driven directly forward are shown in Figure 17. The odom frame has moved forward, while the rest of the transforms are stationary, located near the world origin. The transforms show that the kinect_link does not move with the odom frame.

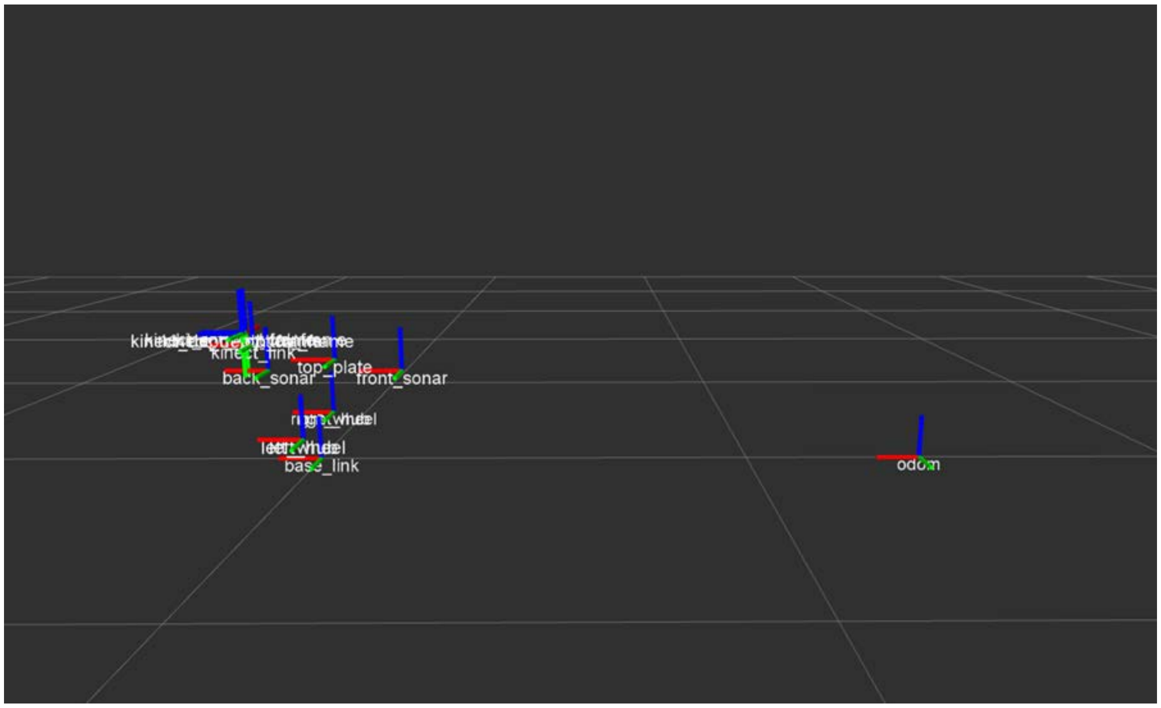

Figure 17. Mobile Robot Transforms

The openni_tracker node using NITE expects the Kinect sensor to be stable and unmoving. With a written in saturation only allowing tracking up to 4.0 m away from the sensor, if the kinect_link is used as the parent frame, the accuracy of the tracker fails 4.0 m from the start location because it gives positions relative to the stationary kinect_link located near the world origin. As such, it was necessary to set the camera_frame_id for the node to the moving odometry frame of the robot, the odom frame, which is constantly changing. The command line to make this change is shown in Figure 14. This ensures that the data being published by the openni_tracker node is relative to the location of the mobile robot.

The openni_tracker callback function is a simple function which may be viewed in Appendix B. The function takes in the transform message as an input. It extracts the ChildFrameId of the message, which contains the joint name. As shown in the example message in Figure 15, the ChildFrameId has a number appended on the end. As this number can change if the user has to recalibrate during testing, only the first five characters of the string are checked. The position of the transform relative to the odometry frame is extracted, and the goal position is set to the x and y location of the transform

$$Goal_{rel} = \begin{bmatrix} x \\ y \end{bmatrix}. \tag{1}$$

A global counting variable for the skeleton tracking is set to zero each time a new skeleton message is received with the torso frame.

Although gesture recognition is not studied in this thesis research, code was included in the callback function to determine the angle of the transform frame. The quaternion message data was extracted and converted to Euler angles. The Euler angles, as well as the difference between the past angles and the current angles, were stored in global variables to allow for further post analysis of the rotations of the transform frames if desired.

C. KINECT DEPTH CALLBACK

The second callback ROS subscriber function was written to analyze the depth data from the Microsoft Kinect sensor. As discussed in Chapter III, the openni_launch package launches nodes to process the data streams from the sensor and convert that data into usable message types. The nodes output a large number of ROS topics from which the user can choose the data that best works for their needs. Because the relative position between the person and the mobile robot was the desired information, it was decided that the raw depth data from the sensor should be used. With the expectation of a noisy indoor environment, image segmentation was run within the callback to determine if a person existed. The callback function MATLAB script is located in Appendix C.

The first step in analysis was reading the image in the message using MATLAB commands. Any point in the image at a distance greater than 4.0 m was set to zero. The documentation for the Microsoft Kinect states that tracking is optimal up to 3.5 m, but the openni_tracker node using NITE allowed for tracking up to 4.0 m. The further distance was chosen to allow for the person to be further from the robot during motion. Next, the data in rows below a specific threshold was set to zero to remove the data points from the ground. The threshold was determined through experimentation. An example of a depth image before and after the background data is removed is shown in Figure 18. Comparing the depth images, we see that the removal of depth pixels past 4.0 m removes the background noise in the image, specifically the ceiling data. Removing the data points from the bottom of the image removes the ground data picked up by the sensor.

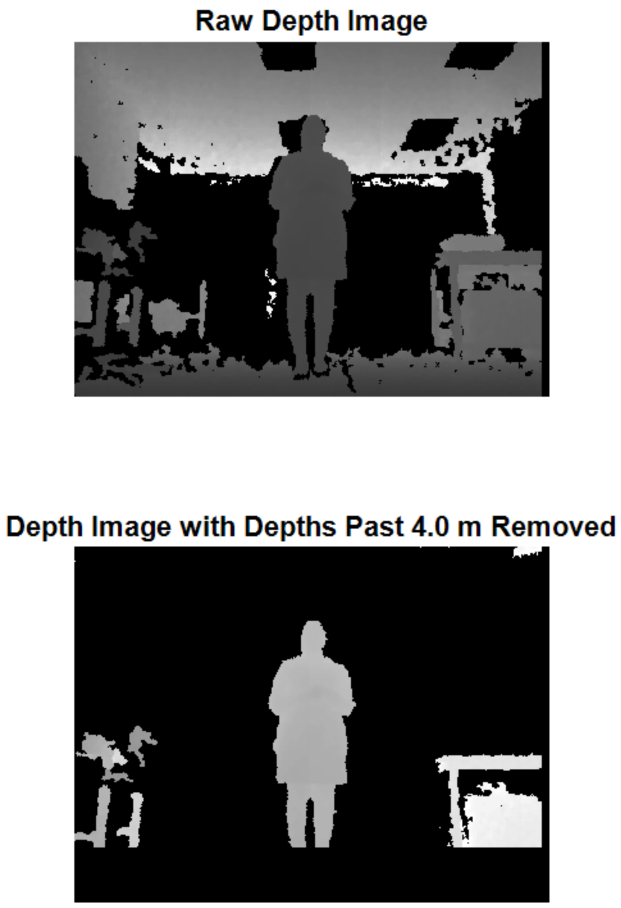

Figure 18. 640×480 Depth Images

As discussed in the prior works section of Chapter I, research has been conducted into using histograms of oriented depth (HOD) image methods on the Microsoft Kinect data. These papers, specifically a paper that investigated image segmentation from histogram of depth data by Dinh et al., motivated this section of the research [54]. The histogram of the data was calculated with a set number of bins. An example histogram of the data is shown in Figure 19. The top subplot shows the histogram of the raw data before any background portions of the image are removed, and the bottom subplot shows the histogram of the data once the background and floor noise were removed. The histograms allow us to see the depths at which a large number of pixels are concentrated.

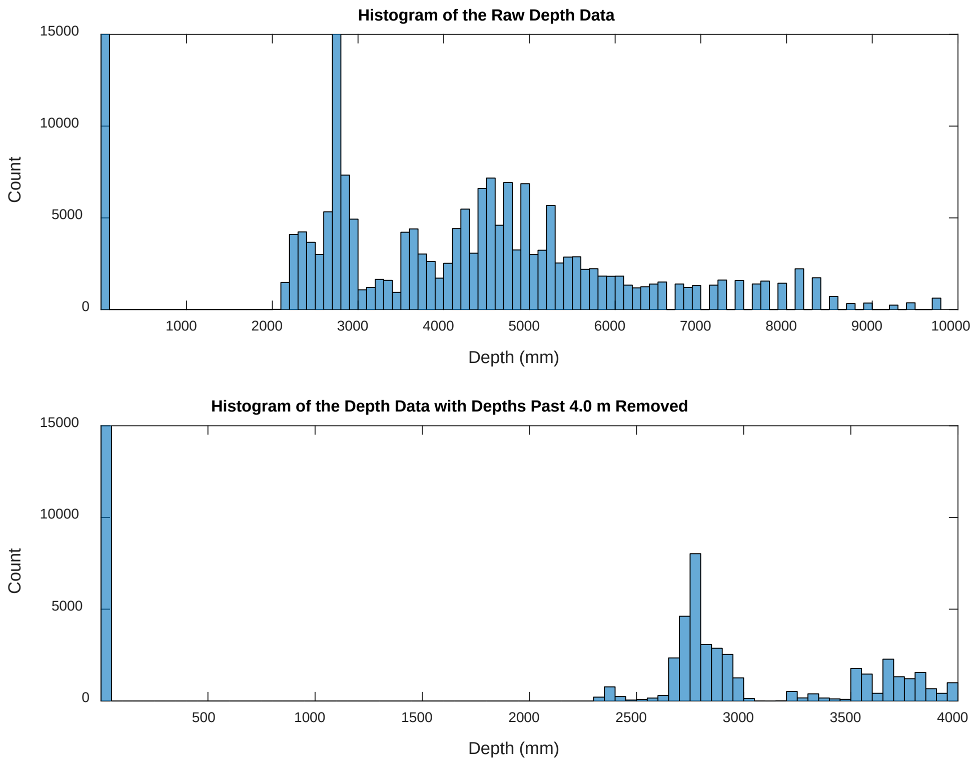

Figure 19. Histograms of the Depth Data

The number of data points in three sequential bins was calculated and compared to a thresholding value determined by experiment. If the number of pixels within the three bins combined was greater than the threshold, it was determined that the average distance of those bins was a region-of-interest in the image where a large amount of data resided. Then the process was repeated for all bins of the histogram to detect which other depths were important. The first bin was ignored because it contained all the zeros where no data existed for the depth. Some of the determined regions-of-interest were very close in value which was not ideal.

Computational load makes it desirable that there be as few regions-of-interest as possible. Dinh et al. in their research used a threshold that was the difference between the nearest and furthest depth of the human body to help determine the regions-of-interest [54]. In their paper, they used a threshold of 0.4 m and compared it to a vector of detected peaks. For the image segmentation in this callback function, detected peaks were not used. Instead, the thresholding message described previously was used to determine regions-of-interest. With a threshold of 0.2 m, half of the threshold used by Dinh et al., led to better results. If two depth regions-of-interest fell within 0.2 m of each other, the two depths were averaged. By combining these two methods, we see that the regions-of-interest for the depth image were determined.

For each of the regions-of-interest, image segmentation was run on the depth data contained in the image within a range of that distance. Image segmentation began with the smallest value depth region-of-interest. A lower and upper distance bound were determined to include the data within 0.35 m of the determined depth. The depth image was then converted to a binary image with the data in the region set to one and all other depth data not contained in that region set to zero. An example binary image for a region-of-interest can be seen in Figure 20. The binary image only takes the data within a set range of the region-of-interest. With a binary image, the image processing toolbox functions in MATLAB can be utilized.

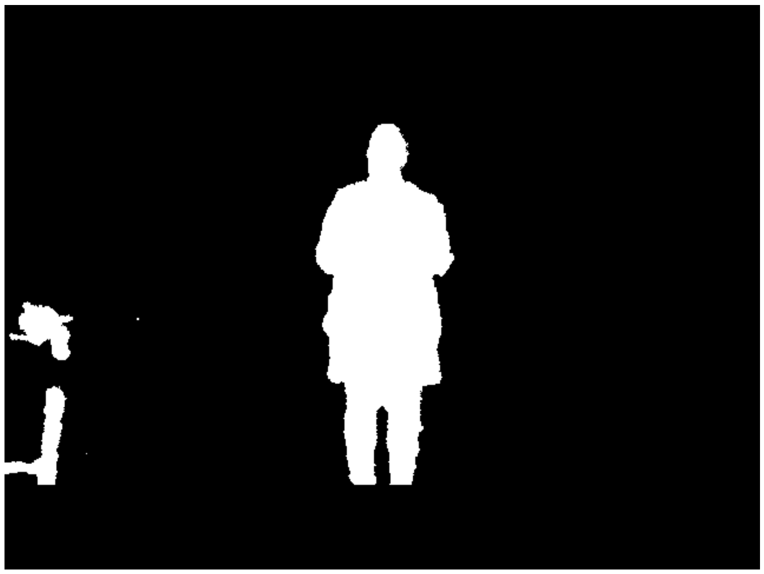

Figure 20. Binary Image for Depth Region-of-Interest

The image processing toolbox has multiple functions that can be utilized to analyze a binary image. First, the bwconncomp command determined the connected components of the binary image. This function was chosen because of its low memory use compared to other functions. The output is a structure which contains the number of objects and the number of pixels within each of those objects. For each object determined, the number of pixels in the object was compared to a threshold. If the number of pixels was less than the threshold, the object was set to zero. This removes any small components in the image not part of the larger objects. Once the small components were removed, the connected components was again calculated for the resulting image. An example of the binary image after the small objects were removed can be seen in Figure 21. By comparing Figure 20 to Figure 21, we can see that only the large object in the region still exists.

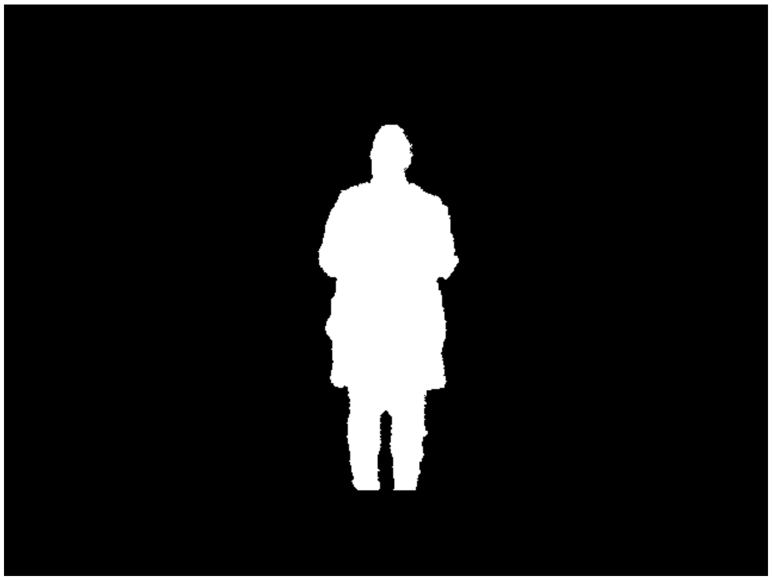

Figure 21. Example Resulting Image Segmentation

Next, the regionprops command was utilized to measure properties of image regions. The properties that were returned included the Area, Centroid, Orientation, BoundingBox, and EulerNumber of each connected component. The Area is simply a scalar specifying the number of pixels in the region. The Centroid of the object is a vector containing the x and y coordinates of the centroid of the connected components. The Orientation is a scalar value that specifies the angle between the x-axis and the major axis of an ellipse that contains the region. The BoundingBox is the smallest rectangle containing the connected components. It specifies the upper-left corner point of the rectangle as well as its length and width in pixels. Lastly, the EulerNumber is a scalar value that specifies the number of holes in the object. Each of these parameters were used to determine whether the object at the depth was a person. As it was possible for multiple objects to exist in the region-of-interest, each connected component was considered a separate object. The properties of each object were compared to thresholds for the parameters that were determined experimentally. The experimentation to determine the thresholds as well as the resulting thresholding values used are discussed in the results section of Chapter V.

If it was determined that the object in the binary image was a person, its location had to be determined. Once a person had been determined to be in the image, the image segmentation process loop stops. By taking the y-centroid of the object, we determine the angle between the center of the image and the person. First, the distance in pixels from the center of the image is determined by subtracting 320 pixels from the y-centroid value. Then the distance in pixels must be converted to meters using a conversion factor determined experimentally. With the depth region-of-interest known and the distance in y from the center of the image, the relative angle between the Microsoft Kinect located onboard the mobile robot and the person was determined, as shown in Figure 22.

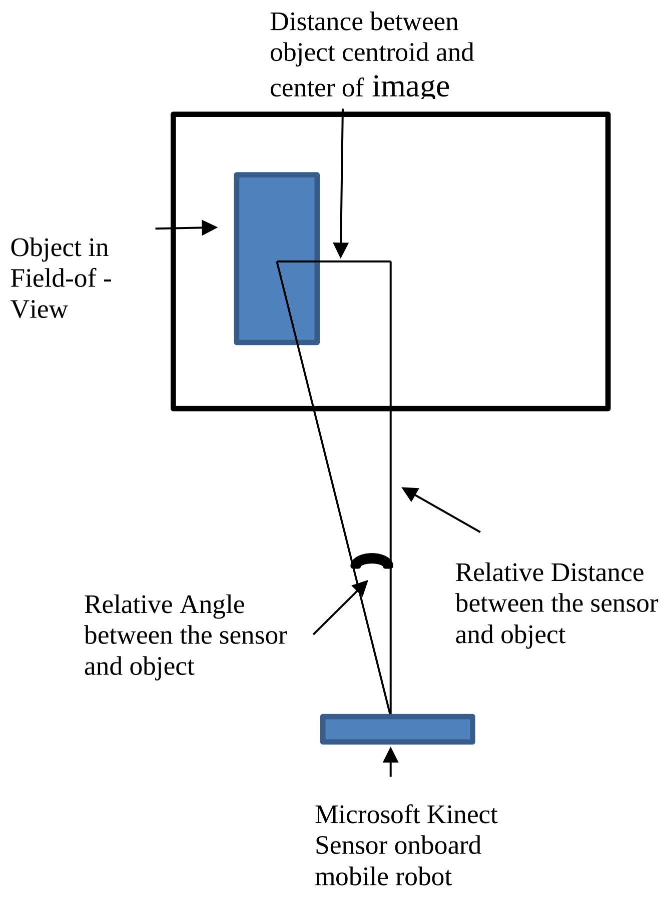
Distance between object centroid and center of image
Object in Field-of - View
Relative Angle between the sensor and object
Relative Distance between the sensor and object
Microsoft Kinect Sensor onboard mobile robot

Figure 22. Explanation of Calculation of Relative Angle between the Sensor and Object in Image

During the real-time running of the callback function, we did not plot the images as this slowed down the running of the script; however, the callback function was rewritten into a script that could be run post testing using a rosbag file recorded during testing. The rosbag file contained the Microsoft Kinect depth data. During the post processing, we plotted the depth images, histograms, and binary images for analysis. If a person was found, the centroid and bounding box around the person were also plotted. The figures shown in this section were all created during post processing to visualize the actions taken by the algorithm in real time. The post-processing script is found in Appendix D.

D. P3-DX ROS SUBSCRIBER CALLBACK FUNCTIONS AND ROS PUBLISHER

Two actions were taken in MATLAB pertaining to the P3-DX mobile robot. First, a ROS publisher had to be set up to be able to send velocity commands across a topic back to the RosAria node. The topic on which these commanded velocities were published is the /my_p3dx/cmd_vel. A ROS publisher and a corresponding rosmessage were created. The RosAria node, as described in Chapter III, subscribes to this topic and uses it to set velocity commands for the robot in ARIA. The rosmessage is where the velocities can be set. With the send command, velocity messages can be published to the ROS Network on the topic. The second action that had to be taken in MATLAB was writing a ROS subscriber callback function to run each time a new robot pose topic was published. The topic was published on the /my_p3dx/pose by the RosAria node. The callback function was passed the ROS publisher and the rosmessage as arguments to allow the velocity to be set within the callback. The callback functions for the mobile robot can be found in Appendix E.

For each new pose data received, the message had to be processed. Similar to the skeleton transform data, when received in MATLAB, the pose message type has a specific object structure. From this structure, the X and Y position, as well as the orientation of the robot, had to be extracted. The X and Y positions of the P3-DX were also stored in a ROS Vector3 containing the X, Y, and Z position of the robot. The Z position of the robot was ignored. The orientation of the P3-DX was stored as a ROS Quaternion. To determine the heading of the robot, the quaternion was converted to Euler angles. The first of the three Euler angles pertains to the heading. The x-position, y-position, and heading of the robot θ_r were stored in a variable

$$Pose = \begin{bmatrix} x \\ y \\ \theta_r \end{bmatrix}. \qquad (2)$$

The relative goal position of the person being tracked was passed to the odometry callback function through the use of global variables. If the callback for the tracker was being used, the goal positions were given directly as x and y relative positions. If the callback for the raw depth data was being used, the goal position was passed as a relative depth and relative angle to the person. Due to the small difference in these two forms of the relative goal data being passed, two separate callback functions were written for the mobile robot to process the data; however, once the global goal position was processed, the two functions were identical.

When the tracker callback was running, the relative x and y position of the person was passed using the global variables. Before calculating the absolute position of the target, it was checked if a new goal or target position had been received. Each time a new position was passed, the absolute target position was calculated by adding the relative position as defined in (1) to the robots position as

$$Goal = \begin{bmatrix} Pose(1) \\ Pose(2) \end{bmatrix} + Goal_{rel}. \tag{3}$$

This calculated goal position was now in absolute world coordinates. If no new goal was passed from the tracker, the past goal position within the current callback was used.

When the raw depth callback was running, the relative angle and relative distance between the target and the robot were passed as global variables. Like the first method, before a position was calculated, it was first determined if new relative data had been passed. If new relative data had been passed, using coordinate transforms, we calculated the relative position of the target. The coordinate transform for the mobile robot is a rotation in the z-axis, as

$$R_{robot} = \begin{bmatrix} \cos(\theta_r) & -\sin(\theta_r) \\ \sin(\theta_r) & \cos(\theta_r) \end{bmatrix} \tag{4}$$

with θ_r the heading of the mobile robot. The coordinate transform for the target is also a rotation in the z-axis,

$$R_{target} = \begin{bmatrix} \cos(\theta_t) & -\sin(\theta_t) \\ \sin(\theta_t) & \cos(\theta_t) \end{bmatrix} \tag{5}$$

with θ_t the relative angle passed from the depth callback. Lastly, the relative distance is written as a 2×1 column vector

$$D_{rel} = \begin{bmatrix} d_{rel} \\ 0 \end{bmatrix} \tag{6}$$

with d_{rel} the relative distance to the target passed from the depth callback. The relative position P_{rel} of the target is calculated using

$$P_{rel} = R_{robot} R_{target} D_{rel} \tag{7}$$

by substituting (4), (5), and (6) into (7). With the relative position calculated, the absolute position of the target in the world frame can be found by adding the relative position to the position of the robot using (3) by substituting (7) for $Goal_{rel}$. If no new relative data was passed from the depth callback, the past absolute goal position was used.

Although the method for calculating the absolute target position was slightly different, the two callbacks functioned in exactly the same way after that process. With a goal target position and the position of the robot, the distance error, defined as the distance to the target, was calculated as

$$derror = norm\left(Goal - \begin{bmatrix} Pose(1) \\ Pose(2) \end{bmatrix} \right). \tag{8}$$

The angle between the robot's position and the goal position was also determined as

$$theta\_g = \arctan\left(\frac{Goal(2) - Pose(2)}{Goal(1) - Pose(1)} \right). \tag{9}$$

The heading of the robot was subtracted from this angle to determine the angular error as

$$ang\_error = theta\_g - \theta_r, \tag{10}$$

which is the error between the robot's current heading and the heading needed to drive directly towards the target.

The next step was to determine if the robot should turn or drive forward. First, it was checked that the angular error was in the range of $[-\pi, +\pi]$. If the angular error determined was outside of this range, 2π was added or subtracted from it to place it in the range. The robot was treated as a differential drive robot, and as such, it was only allowed to turn or drive forward. This was to allow for easier understanding of the actions of the robot during testing while avoiding excess jostling of the sensor. The maximum translational and rotational speed of the robot was set to 0.2 m/s and 0.2 rad/s, respectively. If the absolute angular error was greater than a set value, the angular velocity of the robot was set using a proportional gain as

$$ang\_vel = K_{p-ang} ang\_error. \tag{11}$$

The gain K_{p-ang} was set to 0.4. The goal translational velocity was set to zero. If the angular error was less than the set value, the distance error was next checked. The actions of the robot fell into three possible options: the robot could drive forward, the robot could stay in place, or the robot could drive backwards. The distances and the actions to be taken are as shown in Figure 23. For straight motion, the translational velocity of the robot was set using a proportional gain as

$$trans\_vel = K_{p-dist} derror \tag{12}$$

with the gain K_{p-dist} set to 0.4 and the angular velocity set to zero.

```
If derror < 1.7 m
        Drive backward slowly
if derror <1.85 and derror>1.7
        Stay in place
else
        Drive forward using proportional gain controller
end
```

Figure 23. Distance Control Actions for P3-DX

Sending the calculated translational and angular velocities without any filtering led to a jerky motion of the robot which rattled the Microsoft Kinect sensor onboard. As such, a low pass filter was used to create a weighted average of the past velocity command and the new command velocity as

$$V_{out} = AV_{cmd} + (1-A)V_{cmd\_past} \tag{13}$$

with V_{cmd} the stacked translational and angular velocity commands, V_{cmd_past} the last velocity commands sent to the robot, and A a weighting coefficient. With the use of the filter, the motion of the robot was weighted towards the past velocity commands with a small value for A, which led to smoother motion of the robot as it changed speeds.

E. SUMMARY

The callback functions described within this chapter allow the control laws to be implemented and the processing of data to be conducted within MATLAB. The callback function for the skeleton tracking was very simple because the x and y relative positions of the person are readily available, and very little processing needs to be done. The callback function for the raw depth information from the Microsoft Kinect is much more complex as image segmentation had to be run to determine if a person existed in the depth image. Many of the parameters used within the callback function were determined through experimentation and are discussed as a part of the results section. Lastly, the callback function for the P3-DX contained the control laws to drive the robot to follow the person found using either of the other two callback functions. Parameters in this callback function were also determined through experimentation and are discussed further as part of the results section. In the results section, we also explain how all the callback functions described were used together to produce the best final successful product.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESULTS

The results of this thesis are discussed within this chapter. The openni_tracker node and its results are discussed with its effects on the parameters in the robot controller explained. Next, the experimentation and methods to determine the parameters for the image segmentation are discussed. Then, the combination of the callback functions into the experiment script file is explained. Lastly, results from the final experiments are discussed.

A. SKELETON TRACKING

The skeleton tracking callback function in MATLAB and the ROS subscriber and ROS publisher for the P3-DX were written into a simple MATLAB script. We wanted to see if the robot could follow a person simply using the position from the skeleton tracking and the current position of the robot. This would allow us to determine how robust the skeleton tracker was during robot motion. With calibration completed, the robot attempted to move towards the goal position where the person was located; however, we saw multiple problems.

The first issue we saw was that when the initial distance between the robot and person was large, the initial velocity command sent accelerated the robot too quickly. The tracking would immediately fail and report that the user tracking information had been lost. The tracking algorithm used from NITE in the openni_tracker node was written for a sensor that was stationary on a platform with the only motion occurring in the person it was tracking. It was determined that the failure seen was caused by the abrupt acceleration of the robot on which the sensor was mounted. The sensor's base is mounted to the built up frame connected to the robot; however, the joint of the sensor that allows it to be tilted was not stabilized. The acceleration of the robot led to a large amount of motion in the horizontal bar of the sensor where the camera sensor and depth sensor are located. As such, we needed to find a way to stabilize the sensor.

Originally, a 3-D printed bracket was used to stabilize the base of the sensor and mount it to the frame added to the P3-DX to hold the sensor and computer. Rather than change the mount itself, dense foam was added under each side of the horizontal bar between the frame and the bar as shown in Figure 24. The simple skeleton tracking test was again run.

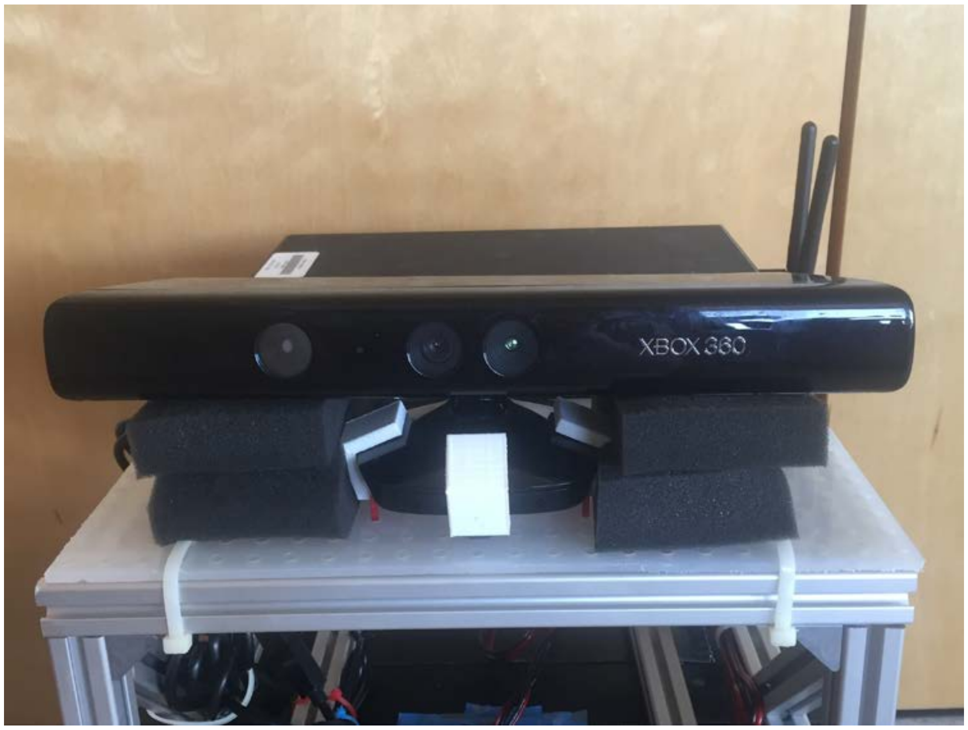

Figure 24. Microsoft Kinect Sensor with Foam Pieces Added for Stability

The addition of the foam below the bar did help decrease some of the motion of the sensor; however, when the robot attempted to move towards the person’s position, the tracking node still failed and said the user tracking information had been lost. We determined that some sort of filter should be added to the motion control laws within the ROS subscriber. The filter explained in Equation (13) was implemented. An initial value for the filter constant A was set to 0.2 to heavily weight the past velocity command sent to the robot. At the start of the experiment, this past velocity command was set to zero. The maximum translational velocity was set to 0.2 m/s, and the maximum rotational velocity was set to 0.2 rad/s.

With the addition of the foam below the sensor bar and the velocity filter, the robot again attempted to follow a person using only the skeleton tracking. This was marginally successful, with the tracker able to track for a few meters and through small amounts of turning. If the robot hit a bump in the floor, sometimes the tracking failed. Other times, the tracking failed simply while the robot was turning. The NITE tracking algorithms do not work well for a person that has turned, with their body and face not directly facing towards the sensor, and it was seen that the tracking failed if the user turned around and walked facing away from the sensor. We determined that the skeleton tracking alone was not adequate for the robot to conduct human following.

Even with the sensor stabilized and the allowed acceleration decreased through the filter, it was seen that as the robot moved forward, and especially when the robot turned, ghost users were found. Ghost users are new users that the node registers while both the robot and the person are in motion. These new users are caused by an introduction of noise from the motion into the data. An example of the terminal print out during the running of the openni_tracker node during robot motion is seen in Figure 25. With a single person in the field-of-view, ghost users are seen by the sensor and lost during robot and human motion.

```
ragreenb@cslab:~/catkin_ws/src/p3dx/launch$ rosrun openni_tracker openni_track
tf:=tf_skeleton
[ INFO] [1492557957.331544419]: New User 1
[ INFO] [1492557958.461030211]: Pose Psi detected for user 1
[ INFO] [1492557958.727864364]: Calibration started for user 1
[ INFO] [1492557959.600426388]: Calibration complete, start tracking user 1
[ INFO] [1492557971.244095615]: New User 2
[ INFO] [1492557971.740070224]: Lost user 2
[ INFO] [1492557973.612380156]: New User 2
[ INFO] [1492557973.746457089]: New User 3
[ INFO] [1492557975.011846443]: Lost user 2
[ INFO] [1492557975.012308237]: Lost user 3
```

Figure 25. Terminal Printout for the openni_tracker During Robot Motion

B. COLLECTION OF KINECT RAW DATA FOR IMAGE SEGMENTATION

In order to run the image segmentation on the depth image, it was necessary to determine parameters for the person. As the amount of space the person takes up within the image is strongly affected by the distance of the person from the sensor, a test was conducted to determine equations for these parameters. We determined that the Bounding Box dimensions, the width and height in pixels of a person, as well as the Area the number of pixels a person contains, would be most affected by change in depth.

The test was run by placing the robot at a specific point in the laboratory and measuring out 4.0 m directly from the robot. Within the range, no other objects were located near the center of the field-of-view. The P3-DX was powered up to give power to the sensor, and the launch file was run to start the processing of the Microsoft Kinect data. The raw depth image messages were recorded in a rosbag file through a command in the terminal window. The author walked backward from a distance of 1.2 m to 4.0 m, repetitively taking a few steps and then stopping. During the collection of data the researcher’s arms were kept close by her sides.

This data was post processed using the MATLAB file described in Chapter IV. The method for determining regions-of-interest in the depth image using the histogram had already been implemented, as well as the removal of small objects that were below a certain threshold number of pixels. From the depth images collected in this test, the objective was to determine equations that described the relationship between Bounding Box width, Bounding Box length, and Area of a person in the field-of-view, as well as to determine if the initial guess thresholds used in both the histogram methods and the removal of small objects were accurate. As such, the Area, Bounding Box dimensions, and depth were collected from the raw depth images in which the researcher had stopped moving.

The first result found from the raw data was related to the distance from which a person could be tracked. Due to the tilt of the sensor and its low height off the floor, the range for tracking that is given in the Microsoft Kinect product information, as well as the NITE documentation, was inaccurate. At a distance closer than 1.7 m, a person was not fully within the field-of-view. This discovery had impacts on the control law for the robot, as it needed to keep a larger distance from the person it was tracking in order to keep them within the field-of-view. This discovery also helped explain errors seen in the skeleton tracking when the robot came too close to the person it was attempting to track.

Information from the data was collected for analysis. The resulting data collected from the post processing of the images are shown in Table 2 with the Bounding Box dimensions and Area listed for each collected depth. The Area and Bounding Box dimensions change by a large amount within the range in which a person can be tracked. As such, it was determined that a curve should be fit to the data with the independent variable as the distance.

Table 2. Data Collected During Testing to Determine Human Parameters

Depth (m)	Area	Bounding Box Width	Bounding Box Height
1.8	40000	161	393
1.9	40700	156	380
2.0	34400	145	369
2.3	30000	134	343
2.5	27000	129	326
2.8	22000	117	305
3.2	16300	101	263
3.5	14000	91	251
3.75	12600	87	239

We next plotted the data to determine what degree polynomial would best fit the data in y. A second order polynomial was chosen over a first or third order polynomial because it provided the least error as shown in Figure 26. The standard deviation of the error in predicting the value for the Area over the 100 data points estimated is shown in Figure 26. A first, second, and third order polynomial were tested.

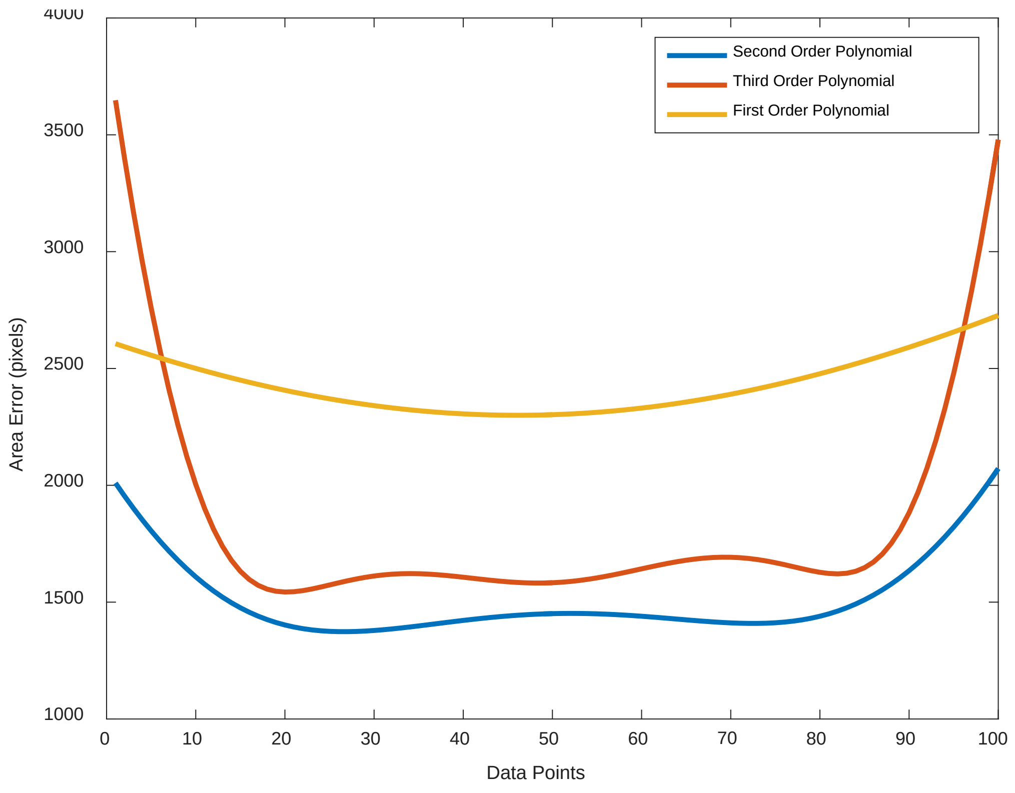

Figure 26. Standard Deviation of the Area Error

Using second order polynomials, we calculated the coefficients of the polynomial. With the coefficients, values for the regression line were calculated and plotted along with the data points. The equations for determining the fit with the data are

$$Area = 4695.4342d^2 - 40595.7005d + 98619.4503, \tag{14}$$

$$BB_{height} = 13.1368d^2 - 152.3431d + 623.3157\ , \tag{15}$$

and

$$BB_{width} = 6.8689d^2 - 75.6528d + 273.226 \tag{16}$$

with d being the relative distance between the sensor and the person. The regression line for the Area is shown in Figure 27, where it can be seen that the Area over the distance range changes by more than 3000 units. The regression line for the Bounding Box height is shown in Figure 28, where it can be seen that the height over the distance range changes by about 200 pixels. Lastly, the regression line for the Bounding Box width is shown in Figure 29, where it can be seen that the width over the distance range changes by about 100 pixels.

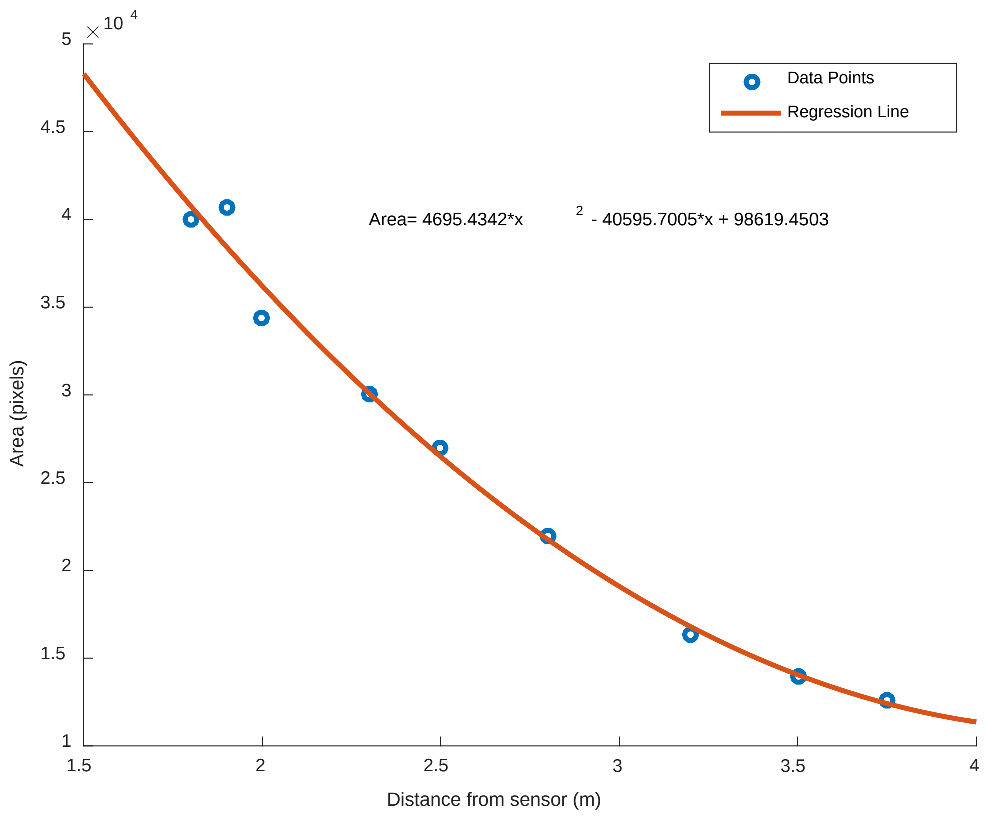

Figure 27. Linear Regression and Data for Area

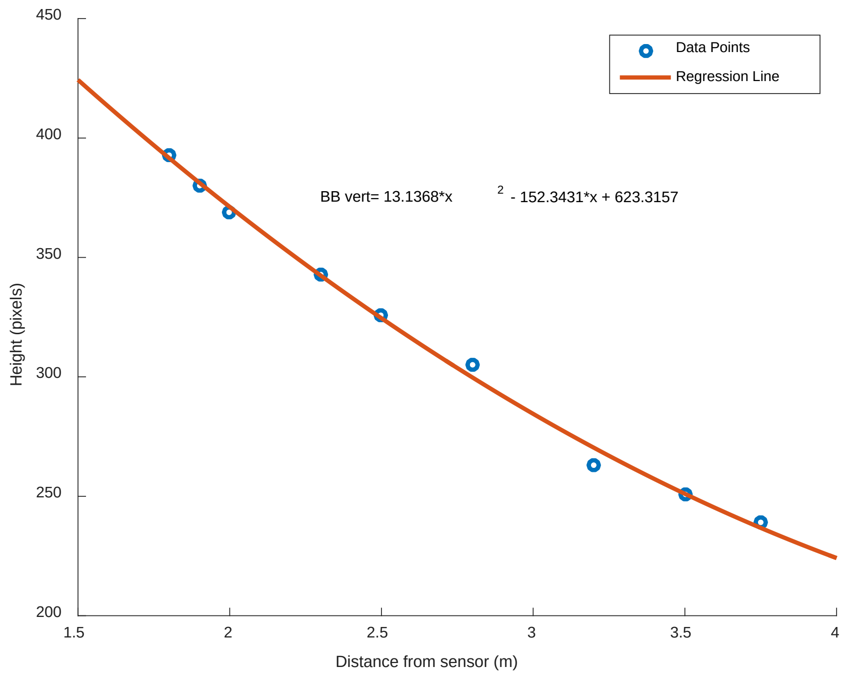

Figure 28. Linear Regression and Data for Bounding Box Height

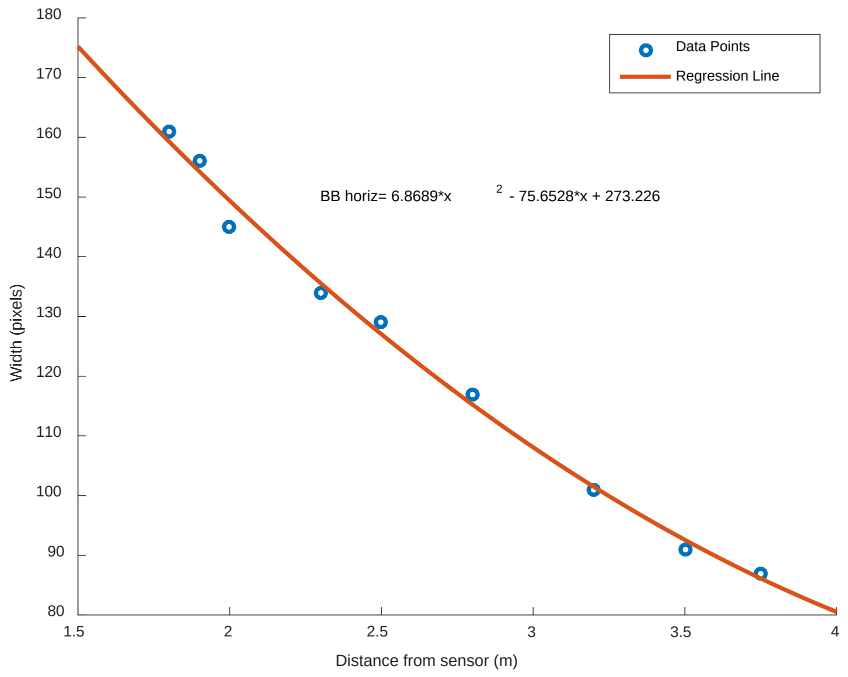

Figure 29. Linear Regression and Data for Bounding Box Width

The other outcome of conducting this test was to determine if the threshold parameters that were guessed would suffice over the entire range in which tracking occurred. Originally, a threshold value of 15,000 pixels was used to determine whether a distance was a region-of-interest; however, from the test it was seen that at a distance of 3.75 m, a person only took up about 12,600 pixels. As such, the threshold was decreased to 10,000 pixels, as it was estimated that a person would take up an area of more than 10,000 pixels even at a maximum tracking distance of 4.0 m. The number of pixels for an object in the binary image to be considered an object and not be deleted was also set to 10,000 pixels.

Five items were checked to determine whether an object was a person: Area, Bounding Box height, Bounding Box width, Orientation, and Euler Number. Using the depth, we calculated the expected Area and Bounding Box dimensions using the regression-line equations. The difference between these parameters and the value calculated for the object were compared to a threshold value. The Orientation of the object relative to the ground was taken and compared to a threshold value for the orientation. Lastly, the Euler Number, the number of holes in the object, was also compared to a threshold value. The threshold values used to determine if an object was a person were initially guessed and updated as data was collected to produce better results. The final threshold values used are shown in Table 3.

Table 3. Threshold Values for Image Segmentation of a Person

Object Parameter	Threshold Value
Area	6000 pixels
Bounding Box height	70 pixels
Bounding Box width	60 pixels
Orientation	75 degrees
Euler Number	10 holes

The large threshold values for the Bounding Box height and width are to allow for different sized individuals as well as different body positions to fit the object parameters. An object had to match all five of the parameters to be determined to be a person and not another object in the field-of-view.

Through the use of the five parameters, it was possible to determine a person from a desk or chair within the environment, as shown in Figure 30. The chosen method of image segmentation was successful in identifying the person in the right side of the image, and the method successfully rejected the laboratory bench which was located in the left side of the image at approximately the same depth.

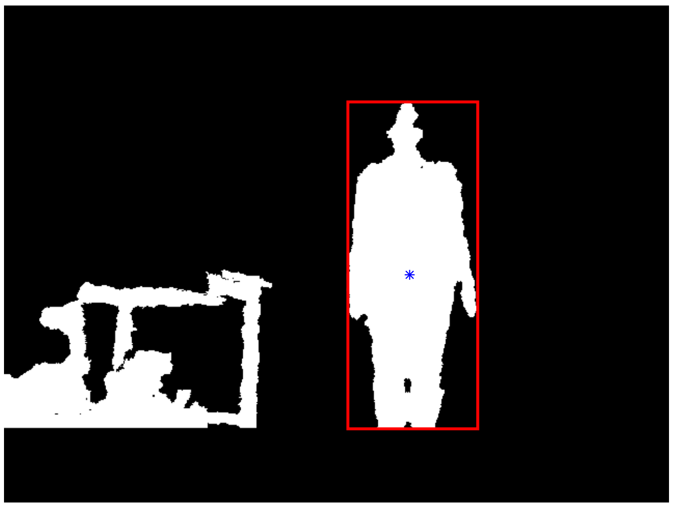

Figure 30. Successful Image Segmentation with Multiple Objects

For testing, only a single person was within the field-of-view; however, we wanted to understand how the algorithm would work with multiple people in the field-of-view. The algorithm looks at each depth region-of-interest separately when running the image segmentation to determine if a person is in the field-of-view. As long as the distance in depth between the two people is large enough, they are determined to be in two separate regions-of-interest. The algorithm starts with the depth closest to the sensor, and so as long as the person meets all the criteria, the closer person to the sensor is set as the goal position.

Data were collected with two people within the field-of-view. The robot was kept stationary, and the two people stood within the field-of-view separated by a great enough distance to not be seen as one object. This test also allowed for the robustness of the threshold values to be checked, as the two individuals were of different heights. At first, the two people stood at different depths, 2.4 m and 3.5 m. The raw depth image, as well as an image with its background data and the floor having been removed, is shown in Figure 31. From the histograms of the depth data once the background has been removed, shown in Figure 32, it is clear that there are two regions-of-interest where the two individuals are standing. The image segmentation is successful, as seen in Figure 33, and the person is located in the right center portion of the field-of-view because that person is at a closer depth of 2.4 m from the sensor.

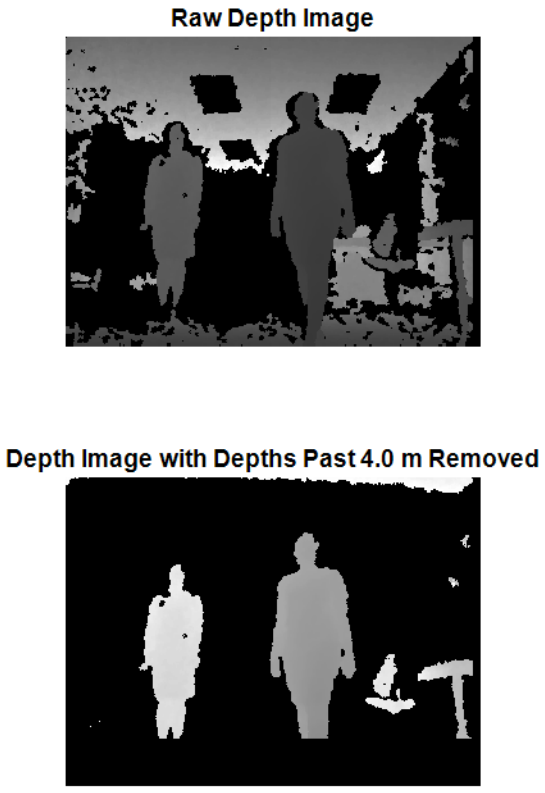

Figure 31. Depth Image during Test with Two People Located in the Field-of-View at Different Depths

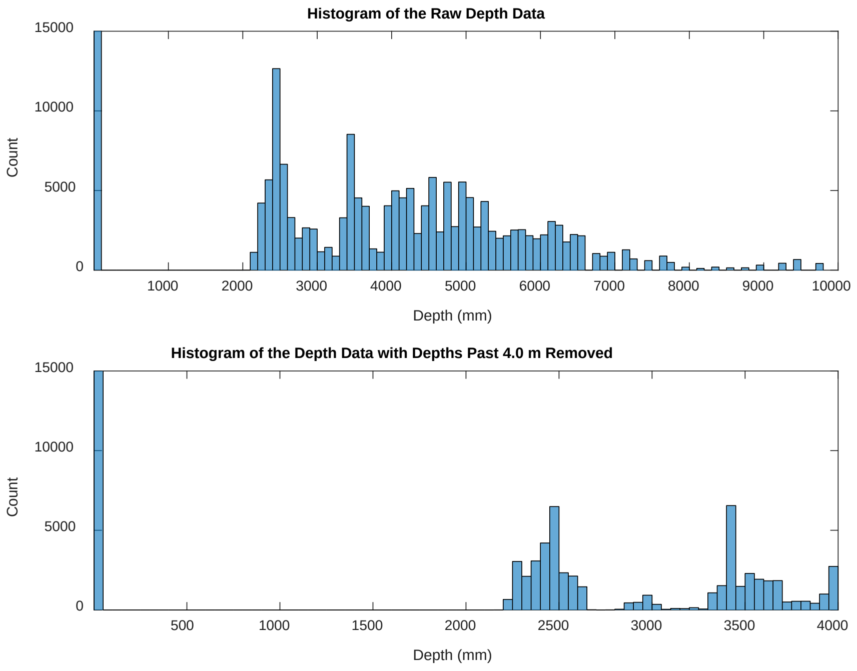

Figure 32. Histogram of Depth Image with Two People in Field-of-View at Different Depths

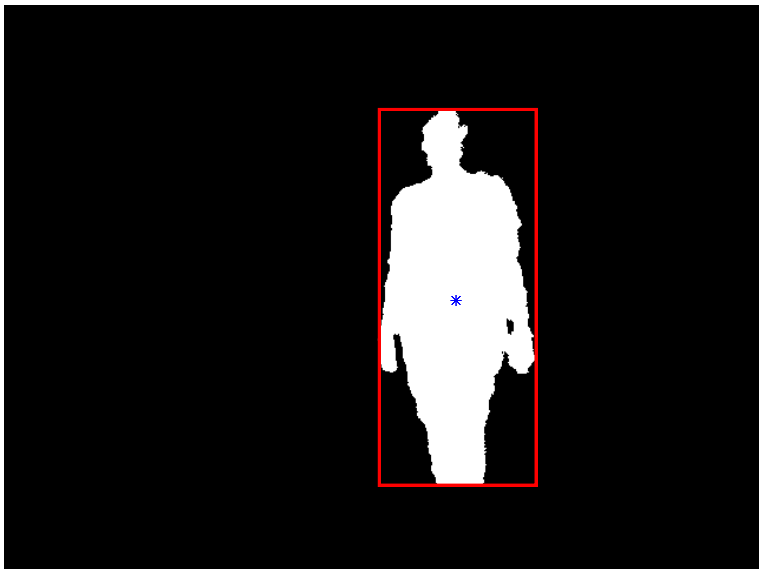

Figure 33. Image Segmentation Result with Two Individuals at Different Depths

Next, the individuals moved and stood at the same distance of 3.4 m to determine how the algorithm would respond to two individuals who fit the parameters at the same depth. Again, the two individuals stood far enough apart that there was no overlap in the image. Looking at Figure 34, we see that other objects exist besides the two individuals in the field-of-view: a chair and part of a laboratory desk. By looking at the histogram in Figure 35 and comparing it to the histogram in Figure 32, we see that there is only one distinct region-of-interest in which a person may be located, such that the two individuals must be located at the same depth.

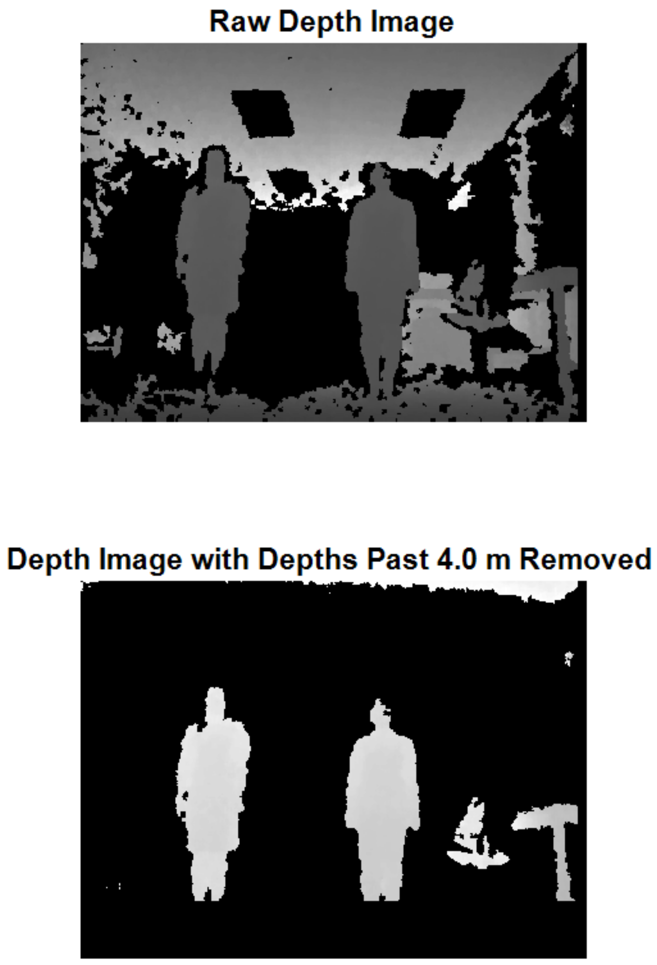

Figure 34. Depth Image During Test with Two People in Field-of-View at the Same Depth

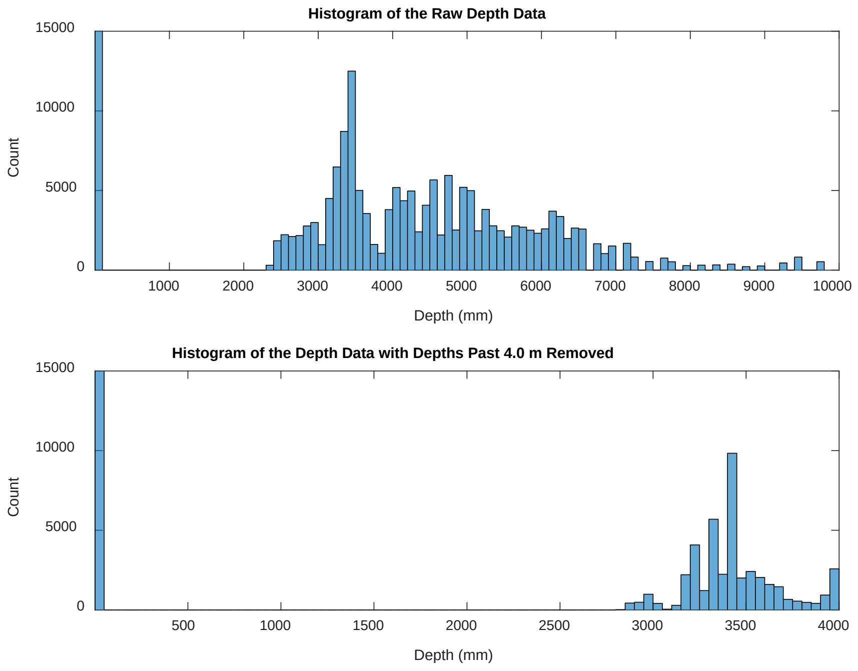

Figure 35. Histogram of Depth Image with Two People in Field-of-View at the Same Depth

In this scenario, the algorithm selects the left person in the image as the goal location. By increasing pixel value location, we see that the objects are sorted in the image segmentation from left to right in the image. Because the algorithm breaks out of the image segmentation loop as soon as a person and its centroid are found, the first person that the algorithm finds, the left individual in this case, is set as the goal position as shown by the rectangle and centroid location overlaid on the depth image in Figure 36. The algorithm succeeds in finding a person even when two individuals are located within the region-of-interest in the image; however, if the other person, the person on the right in this case, was the desired person for the mobile robot to follow, this method fails and picks the wrong user.

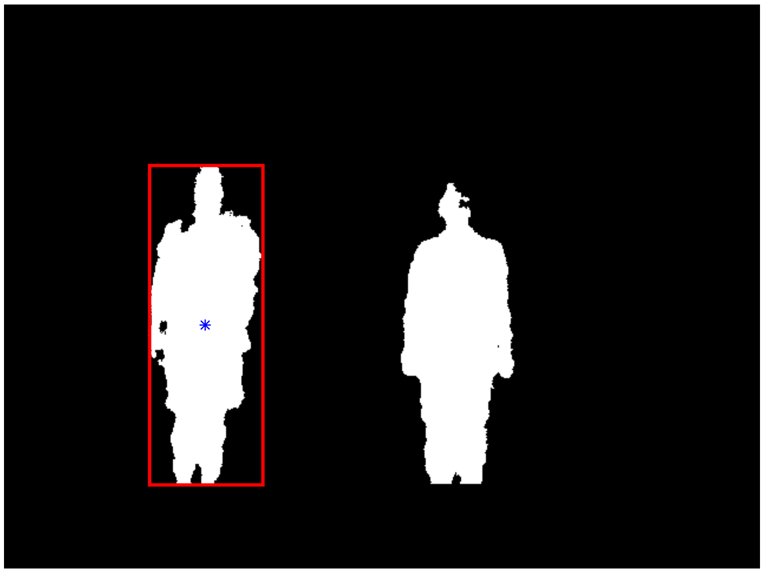

Figure 36. Image Segmentation Result with Two Individuals at the Same Depth

C. FINAL TEST

The final test was conducted within the laboratory environment. The objective was to determine if the robot could successfully follow a person around the indoor environment. Only one person was located standing within the field-of-view of the sensor. During the test, the person walked facing the robot with other objects located in the peripherals.

1. Script File

With a working image segmentation algorithm, the next step was to determine how to combine the skeleton tracking and the image segmentation of the depth image to produce the final result of conducting human following in the environment. As discussed in Chapter I, multiple researchers have looked into using the skeleton tracking as the main tracker and using an auxiliary tracker when the skeleton tracking became unstable. It was decided that this method should be used, with the auxiliary tracker being the depth image segmentation.

Before the experiment can begin, the ROS nodes must be started. On the desktop computer, the roscore, depthimage_to_laserscan, slam_gmapping, and a rosbag collection command are run by means of the terminal window. Using SSH, we ran the openni_tracker, RosAria, and openni_launch nodes on the computer onboard the robot using the terminal window.

A single script file was created to run the experiment. The script utilized is found in Appendix F. Global variables are set to allow information to be passed between the main script and the four callback functions implemented. The ROS publisher for the P3-DX commanded velocity is initialized. A pause is included to allow the user to move from starting the MATLAB script running on the desktop computer to standing in front of the robot and hitting the psi pose to calibrate the tracking node. The /tf_skeleton topic must be publishing a transform for the test to begin. The ROS subscriber callback function for the robot using the skeleton data is then initialized.

The test is run inside of multiple MATLAB while loops. Counters for the robot and skeleton tracking are used to determine how many pieces of each type of information have been passed without a new piece of data being received. This allows for the algorithm to determine when the tracking node has lost the user it was tracking. When the skeleton tracking has lost the user, the Microsoft Kinect raw depth data callback function is initialized and image segmentation begins. The callback function for the robot using the skeleton data is deleted from the workspace, and the callback function for the robot using the goal position from the image segmentation is initialized. If the person recalibrates and the tracking node begins to publish skeleton transforms again, the skeleton counter is reset. The image segmentation is stopped as well as the P3-DX callback function which is using the raw depth data goal position. The skeleton transform position is again used as the goal position to the respective P3-DX callback function. By using two different callbacks for the P3-DX depending on which tracker is being used, we can run the skeleton tracking callback continuously, and the node can wait for the person to hit the psi pose and recalibrate.

2. Results

Using the script file described above, we conducted tests to determine whether it is possible for the robot to successfully track and follow a person through the laboratory environment. The test was successful, and the robot was able to track the person from the start location forward around a grouping of desks and back to the start location. The scenario geometry, the paths of the person and the robot, can be seen in Figure 37. From the data plotted in Figure 37, it does not look as if the person and robot returned to the start location (0, 0). This error seen in the plot can be directly attributed to the odometry error. Odometry error occurs because encoders on the wheels are used to determine how far the robot has travelled and how much the robot has turned, as well as the pose and orientation of the robot. A large amount of error is introduced into the location of the robot due to the turning conducted by the robot to keep the person in the center of the field-of-view.

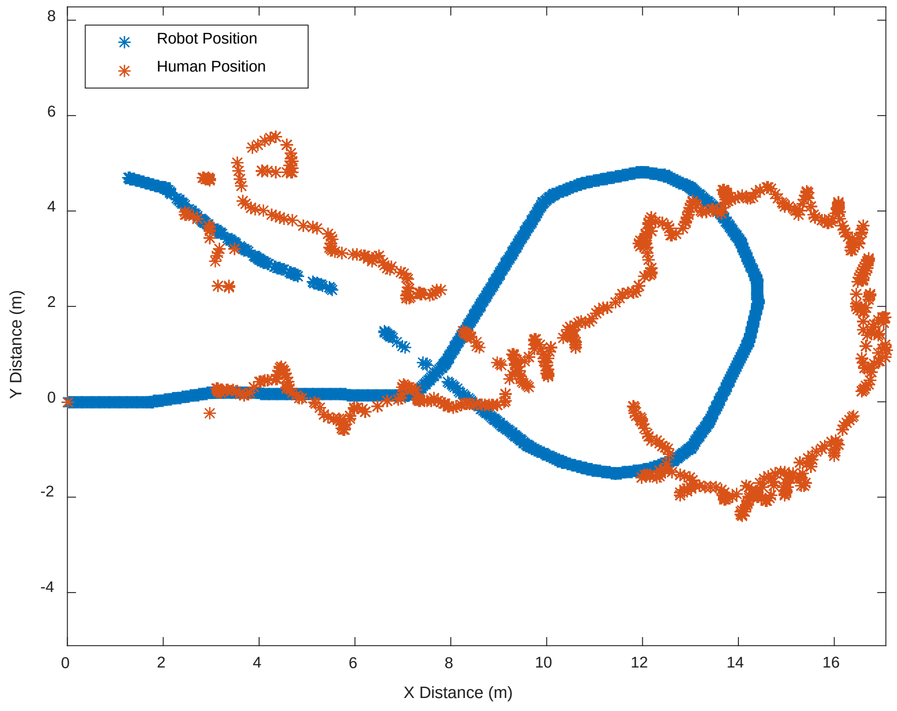

Figure 37. Test Geometry in the XY Plane

The robot’s position in X and Y over the trial can be seen in Figure 38. The robots position was stored for each iteration of the callback function for the robot’s position. As expected, the robot is initially located at the position (0, 0). The goal position, the location of the person being followed in the world frame, is plotted in Figure 39. The goal position of the target was stored for each iteration of the callback function for the robot's position. By comparing Figure 38 with Figure 39, we notice that the path of the robot is much smoother than the path of the person being tracked. This is also evidenced through the geometry plot in Figure 37. This can be attributed to the fact that the robot only has to turn when the person is outside of the allowed angular amount of error. If the person has moved slightly left or right but is still within the allowed angular amount of error, the robot continues to simply travel in a straight line. This can be seen in Figure 37 between an X distance of four to seven meters, as the person walked first to the right and then to the left. The robot's direction only changed slightly through the large motion to the left and right of the person. The lack of smoothness in the position trajectory of the person can also be seen as a result of the tracking. It is believed this was caused by greater error in the position of the person between successive measurements than when compared to the error in the robot's position in successive measurements.

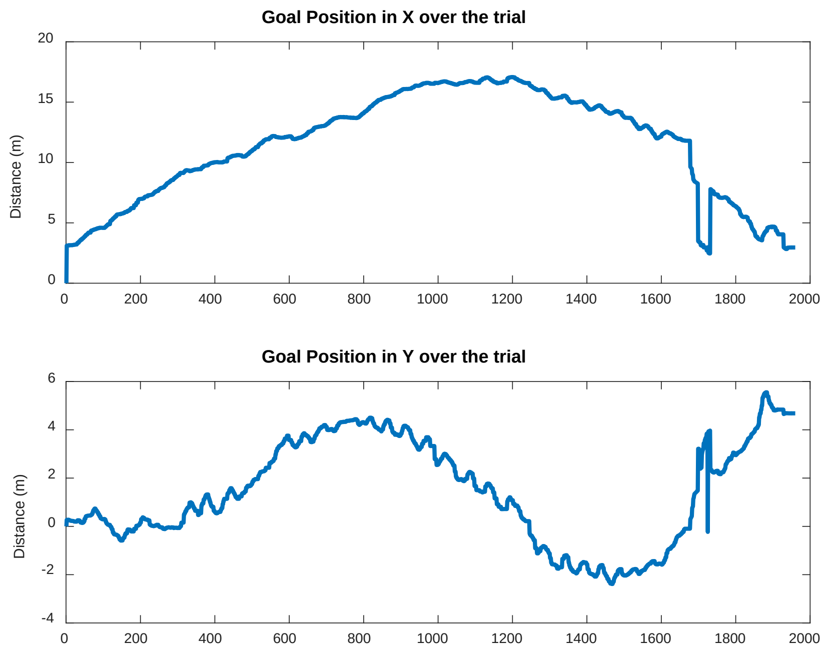

Figure 38. Robot Position over the Trial

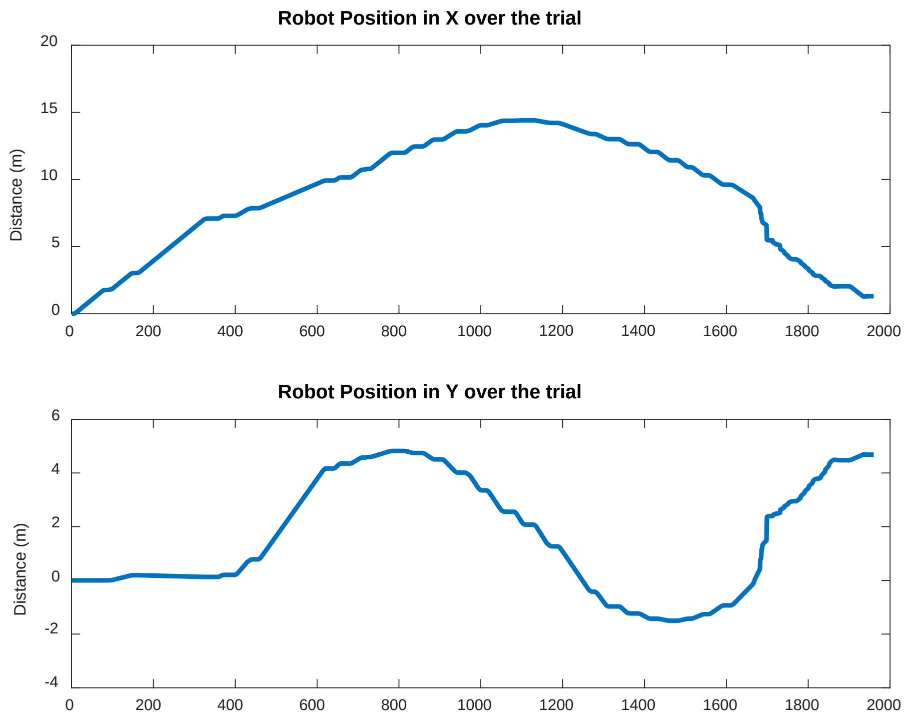

Figure 39. Goal Position over the Trial

The robot's position never reaches the maximums of the goal position. This result was also expected. The robot's task is to keep the person's location within a certain range. As the goal position and position of the robot are constantly updated, it is possible for the robot to travel a shorter path than the person.

Due to the large error in the odometry measurements of the robot, issues were seen in the SLAM creation of the Occupancy Grid. The inputs to the SLAM package are the sensor data and the transforms required for the sensor scans and robot position. Each scan is transformed into the odometry transform frame. Then that information is transformed into a world position. Due to the large amount of odometry error from the angular motions of the robot, large errors are seen in the occupancy grids built from the SLAM. Also, the person constantly moving within the view of the sensor caused problems because the person was seen as an occupied space. Due to these two problems, the map created by the robot as it was tracking a person was deemed unusable.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

A. SUMMARY

Human detection, tracking, and following is one application in which computer vision can be relevant to the field of robotics. In computer vision, human detection and tracking is defined by the objective of finding a human and following the movement of the human using a sequence of images. Original methods of human detection in robotics used a two-dimensional image to estimate a three-dimensional space. This method necessitates estimation as the sensor does not provide enough information; however, with availability of new technology, observation in three dimensions is possible.

The Microsoft Kinect, one of the most successful RGB-D sensors, is known for its human detection capabilities and has multiple software development kits available. With the Microsoft Kinect’s release in 2010, and the release of the Microsoft SDK in 2012, the sensor became widely used in research due to its low cost, availability, and software capabilities. With a camera sensor and depth sensor, the Microsoft Kinect allows for a three-dimensional understanding of the environment.

The objective of this thesis research was to determine if it was feasible to implement human tracking and following on a mobile robot in an indoor environment. Specifically, we wanted to conduct tracking with the Microsoft Kinect using a specific software development environment, ROS and MATLAB. The P3-DX, a Pioneer indoor mobile robot, was chosen to be worked with in this thesis research.

First, the hardware and software for the system had to be set up. Available ROS packages were utilized to run the drivers for the robot and the Microsoft Kinect. MATLAB was utilized to receive and send messages to and from the ROS system. This allowed for the algorithms to be written in MATLAB as callback functions and for experiments to be initialized using a script in MATLAB.

The skeleton tracking capabilities of the Microsoft Kinect, specifically the NITE software, were utilized as the main tracking system. An auxiliary method was created using histograms of depth and region properties to segment a person from the depth image produced by the sensor. From each of these two methods, the goal position of the human was found relative to the mobile robot. This relative position was used to drive the mobile robot towards the location of the person being tracked while keeping the robot a minimum distance from the person to keep them within the field-of-view. A simple control law and low pass filter were utilized to drive the mobile robot to the goal location.

The first result of this thesis research is a successful image segmentation algorithm. A relationship was found between the depth, the independent variable, and the dependent variables Area, Bounding Box width, and Bounding Box height. Using these relationships, as well as other object parameters, we can determine if a person exists in the depth image. This result supported the final objective of conducting human following with a mobile robot.

The P3-DX indoor mobile robot was able to successfully track and follow a person through the indoor environment. Employing ROS, we utilized packages to run the P3-DX, Microsoft Kinect, and NITE skeleton tracking. MATLAB was successfully utilized for implementing algorithms as scripts and callback functions. Using the combination of the ROS packages and the MATLAB codes, we met the objective of this thesis research, proving that it is feasible to implement human following on a mobile robot in an indoor environment.

B. FUTURE WORK

Future work that could be conducted in the realm of this thesis research would be to implement a searching algorithm. The tests run were conducted expecting a person to always be within the field-of-view. Implementing a search method, a person outside of the field-of-view could be detected. This could also allow for the robot to find a person who was being tracked and then moved outside of the field-of-view.

Another area that could be explored would be the combination of multiple sensors onboard the mobile robot to conduct obstacle avoidance while tracking a person. Assuming that no obstacle was located between the mobile robot and the person it was tracking in the environment, we did not implement obstacle avoidance in this thesis research. Combining multiple sensors will allow for this capability to be added. For example, the P3-DX contains an array of range-finding sonars which can be incorporated to detect obstacles. Another option is to utilize two Microsoft Kinect sensors to gain a larger picture of the environment.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. ROS LAUNCH FILES

A. ROSARIA LAUNCH FILE

```
<?xml version=“1.0”?>

<launch>


  <arg name=“port” default=“/dev/ttyS0” />


  <node pkg=“rosaria” type=“RosAria” name=“my_p3dx” output=“screen”>


    <param name=“port” value=“$(arg port)” />


  </node>


  <!-- This following lines are taken from the p2os_urdf package-->


  <include file=“$(find p3dx)/launch/upload_pioneer3dx.xml”/>


  <node pkg=“robot_state_publisher” type=“state_publisher” 
name=“robot_state_publisher”>


    <param name=“publish_frequency” type=“double” value=“30.0”/>


    <param name=“tf_prefix” type=“string” value=““/>


  </node>


  <node pkg=“p2os_urdf” type=“p2os_publisher_3dx” name=“publisher”/>


</launch>
```

B. OPENNI_LAUNCH LAUNCH FILE

```
<?xml version=“1.0”?>

<launch>
```

```
  <!-- setting camera in this file was not working so reset it in
openni.launch to kinect -->

               <include file =“$(find
openni_launch)/launch/openni.launch”/>

               <param name=“respawn” value=“true”/>

               <arg name=“camera” value=“kinect”/>

               <param name=“depth registration” value=“false” />



</launch>
```

C. DEPTHIMAGE_TO_LASERSCAN LAUNCH FILE

```
<?xml version=“1.0”?>

<launch>

<node pkg=“depthimage_to_laserscan” type=“depthimage_to_laserscan”
name=“depthimage_to_laserscan” respawn=“true” >

               <remap from=“image” to=“/kinect/depth/image_rect_raw”/>


               <param name=“scan_height” value=“200”/>

               <param name=“scan_time” value=“0.125” />

               <param name=“range_min” value=“0.45”/>

               <param name=“range_max” value=“7.0” />

               <param name=“min_height” value=“0.05”/>

               <param name=“max_height” value=“1.0” />

               <param name=“output_frame_id”
value=“kinect_depth_frame”/>

</node>

</launch>
```

D. SLAM GMAPPING LAUNCH FILE

```
<?xml version=“1.0”?>

<launch>
```

```
<node pkg="gmapping" name= "slam_gmapping" type="slam_gmapping">
    <param name="maxUrange" value="5.5" />
    <param name="maxRange" value="6.0" />
    <param name="xmin" value="0.0" />
    <param name="xmax" value="20.0" />
    <param name="ymin" value="-10.0" />
    <param name="ymax" value="10.0" />
    <param name="delta" value="0.1" />
    </node>

</launch>
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. OPENNI_TRACKER CALLBACK FUNCTION

```
%tf callback function
function tfCallback(~,message,cmdpub,cmdmsg)

% Skeleton Tracker
% Called when each each new /tf_skeleton message is received

global counter_tf;
global goal;
global angles;
global Translation;
global counter_pose;
global delta_angle
global angle_out
global counter_skeleton
global counter;
goal=goal;
if counter_tf <15
   counter_tf=counter_tf+1;
else
   % Generate a simplified pose
   frame=message.Transforms.ChildFrameId;
   if strncmp(frame,'torso_1',5)
      if counter==0
         %do nothing
      else
         angles_past=angles;
         pose_past=Translation;
      end

      % Extract Pose

Translation=[message.Transforms.Transform.Translation.X,message.Transforms.Transform.Tran
slation.Y,message.Transforms.Transform.Translation.Z];
      quat = message.Transforms.Transform.Rotation;

      % From quaternion to Euler

      angles = quat2eul([quat.W quat.X quat.Y quat.Z]);

      if counter==0
         %do nothing
      else
         delta_angle(counter,:)=angles-angles_past;
         angle_out(counter,:)=angles;
      end
      goal=[Translation(1,1),Translation(1,2)]';
```

```
        counter=counter+1;
        fprintf('Current goal X=%4.2f, Y=%4.2f\n',goal(1),goal(2));
        counter_skeleton=0;
        counter_pose=counter_pose+1
    else
        %     data='not left_elbow';
        return;

    en
    counter_tf=0;

end
```

APPENDIX C. KINECT CALLBACK FUNCTION

```
function kinectdepthCallback(~,message,cmdPub,cmdMsg)

% Callback for depth image raw data editing


%global derror;
global goal_d_rel;
global goal_ang_rel;
global counter_k;
global counter;
% Extract Image Data in message and read into Matlab variable
%Im is 480x640 depth image uint16

Im=readImage(message);
I_show=Im;
I_show = double(I_show)/65535;
% Show image, it's histogram below it
mean_dist=[];

for ii=1:480
        for kk=1:640
        if(I_show(ii,kk)>0.0)
        I_show(ii,kk)=1;
        else
        I_show(ii,kk)=0;
        end
        end
end

for ii=1:480
        for kk=1:640
        if(Im(ii,kk)>4000)
        Im(ii,kk)=0;

        end
        end
end
% Set data coming in in the bottom 70 rows to zero to cut out ground
for ii=410:480
        Im(ii,:)=0;
end

i=h.NumBins-2;
counter=1;
while i>1 %attempting to ignore everything below certain depth in histogram
        %for i=h.NumBins-2:-1:2 %number of bins
        if ((h.Values(i)+h.Values(i+1)+h.Values(i+2))>10000)
        pixelValue=h.Values(i)+h.Values(i+1)+h.Values(i+2);

        dist_b=((h.BinLimits(2)/h.NumBins)*(i+2)+(h.BinLimits(2)/h.NumBins)*(i+1)+(h.BinLi
        mits(2)/h.NumBins)*(i))/3000;

                if dist_b ~= 0
                        dist_h2(counter)=dist_b;
                        counter=counter+1;
                end
        end
        i=i-1;
end
if exist ('dist_b')
        if (dist_b<1.0)
        low_dist=dist_b;
        dist_b=0;
        end
else
        dist_b=0;
        dist_h2=0;
end
count=length(dist_h2);
% combine distances if closer than 0.2 meters
while(length(dist_h2)>1)
if (abs(dist_h2(length(dist_h2))-dist_h2(length(dist_h2)-1))<0.2)
        ans=(dist_h2(length(dist_h2))+dist_h2(length(dist_h2)-1))/2;
        dist_h2(length(dist_h2)-1)=ans;
        dist_h2(length(dist_h2))=[];
else
        mean_dist=[mean_dist,dist_h2(length(dist_h2))]
        dist_h2(length(dist_h2))=[];
end
end
mean_dist=[mean_dist,dist_h2(1)];
%loop through each distance and run image segmentation on each
for pp=1:length(mean_dist)
T=700;
upper=mean_dist(pp)*1000+T/2;
lower=mean_dist(pp)*1000-T/2;
% Convert image from depth image to decimels, then convert to binary using
% for loop
I64 = double(Im)/65535;%imshow(I64)
for ii=1:480
        for kk=1:640
        if(Im(ii,kk)<lower || Im(ii,kk)> upper)
        I64(ii,kk)=0;
        else
        I64(ii,kk)=1;
        end
```

```
	end
end

BW=I64;
Area=bwarea(BW);
CC=bwconncomp(BW);
BW=imfill(BW,8,'holes'); %Doesn't do crap
numPixels = cellfun(@numel,CC.PixelIdxList);
[biggest,idx] = max(numPixels);

for i=1:CC.NumObjects
	if numPixels(i)<10000
	%if the object is smaller than 10000 pixels set it to zero
	BW(CC.PixelIdxList{i}) = 0;
	end
end

% figure()
% imshow(BW)
Area=bwarea(BW);
CC=bwconncomp(BW);
if(Area>10000 && dist_b ~=0)
	s=regionprops(CC,'Area','centroid','MajorAxisLength','MinorAxisLength','Orientation','B
oundingBox','EulerNumber','Perimeter');
	centroid=cat(1,s.Centroid);
	Major=s.MajorAxisLength;
	Minor=s.MinorAxisLength;
	%hold on
	p_A=[4695.43419202680,-40595.7005014340,98619.4503311143];
	p_B1=[6.86887903612834,-75.6528307159344,278.225979474800];
	p_B2=[13.1367701568270,-152.343060807504,623.315704575546];
	Area_r=p_A(1)*dist_b^2+p_A(2)*dist_b+p_A(3);
	B1_r=p_B1(1)*dist_b^2+p_B1(2)*dist_b+p_B1(3);
	B2_r=p_B2(1)*dist_b^2+p_B2(2)*dist_b+p_B2(3);

	error=6000;

	for k = 1 : length(s)
	BB = s(k).BoundingBox;
	Orient=s(k).Orientation;
	AREA=s(k).Area;
	holes=s(k).EulerNumber;
	cent=cat(1,s(k).Centroid)
	if (abs(BB(3)-B1_r)<60 && abs(BB(4)-B2_r)<70 && abs(Orient)>75.0 && abs(AREA-
Area_r)<error && abs(holes)<=10 && abs(cent(2)-240)<100)

	centroids=cat(1,s(k).Centroid);
	else
	end
```

```
            end

else
            ang=0;
end
  if exist('centroids')
            break;
  end
end
% Calculating relative position using distance and centroid
% Image is a 480 x 640 480 tall, 640 wide
% knowing distance we should be able to calculate angle from center
% center is at 320 pixels
if exist ('centroids')
            y_pixels=centroids(1)-320;
            % conversion from pixels to meters, found using experimentation
            %

            if y_pixels<0
            pix_to_meters=0.0065;
            else
            pix_to_meters=0.0055;
            end

            ang=atan2((y_pixels*pix_to_meters),mean_dist(pp));

            %setting global vairables
            goal_d_rel=mean_dist(pp);
            goal_ang_rel=ang*-1;
            counter_k=0;
            counter=0;
            %fprintf('error=%4.2f, angerror=%4.2f\n',goal_d_rel,goal_ang_rel);
else
            % if no person is found in the image
            counter_k=counter_k+1
            goal_d_rel=1.75;
            goal_ang_rel=0;
            % if the smallest distance to an object is less than 1.6, set that distance as the goal
            % to make the robot back up
            if min(mean_dist)<1.6
            goal_d_rel=min(mean_dist)
            if exist ('low_dist')
            goal_d_rel=low_dist;
            goal_ang_rel=0;
            end
            counter=0;
end

end
```

APPENDIX D. KINECT POST-PROCESSING SCRIPT

```
close all; clear all;
%filepath=’fill in name of bagfile.bag';

filepath='trialdata.bag';
bag = rosbag(filepath);
bag.AvailableTopics
%bag.MessageList;
counter_k=0;
%%
bagselect1=select(bag, 'Topic','/kinect/depth/image_raw');
Msgs=bagselect1.MessageList;
msgs2=readMessages(bagselect1,[1:200]);
distance=[];
angles=[];

%%

% The for loop runs through every fifth image and runs the image segmentation and
% plots the images and histograms
for jj=1:5:200
Im=readImage(msgs2{jj});
mean_dist=[];
figure()
subplot(2,2,1)
imshow(Im)
image(Im,'CDataMapping','scaled')
subplot(2,2,3)
histogram(Im)
for ii=1:480
   for kk=1:640
   if(Im(ii,kk)>4000)
       Im(ii,kk)=0;

   end
   end
end
for ii=410:480
   Im(ii,:)=0;
end

subplot(2,2,2)
imshow(Im)
image(Im,'CDataMapping','scaled')

subplot(2,2,4)
histogram(Im,35)
```

```
%%
h=histogram(Im,35);
i=h.NumBins-2;
counter=1;
while(i>1)

   if ((h.Values(i)+h.Values(i+1)+h.Values(i+2))>10000)
       pixelValue=h.Values(i)+h.Values(i+1)+h.Values(i+2);

       dist_b=((h.BinLimits(2)/h.NumBins)*(i+2)+(h.BinLimits(2)/h.NumBins)*(i+1)+(h.BinLimit
s(2)/h.NumBins)*(i))/3000;
       if dist_b ~= 0
           dist_h2(counter)=dist_b;
           counter=counter+1;
       end

   end
   i=i-1;
end

%end
if exist ('dist_b')
   if (dist_b<1.7)
       low_dist=dist_b;
       dist_b=0;
   end
else
   dist_b=0;
   dist_h2=0;
end
distance=[distance,dist_b];
count=length(dist_h2);

while(length(dist_h2)>1)
if (abs(dist_h2(length(dist_h2))-dist_h2(length(dist_h2)-1))<0.2)
   ans=(dist_h2(length(dist_h2))+dist_h2(length(dist_h2)-1))/2;
   dist_h2(length(dist_h2)-1)=ans;
   dist_h2(length(dist_h2))=[];
else
   mean_dist=[mean_dist,dist_h2(length(dist_h2))]
   dist_h2(length(dist_h2))=[];
end
end
mean_dist=[mean_dist,dist_h2(1)]

for pp=1:length(mean_dist)

T=700;
```

```
upper=mean_dist(pp)*1000+T/2;
lower=mean_dist(pp)*1000-T/2;
I64 = zeros(480,640);%imshow(I64)
for ii=1:480
  for kk=1:640
  if(Im(ii,kk)<lower || Im(ii,kk)> upper)
      I64(ii,kk)=0;
  else
      I64(ii,kk)=1;
  end
  end
end
%pause
figure()
imshow(I64)
BW=I64;
Area=bwarea(BW);
BW2=imfill(BW,8,'holes'); %Doesn't do crap
CC=bwconncomp(BW);
numPixels = cellfun(@numel,CC.PixelIdxList);
[biggest,idx] = max(numPixels);
for ii=1:CC.NumObjects
  if numPixels(ii)<10000
      %if the object is smaller than 10000 pixels set it to zero
      BW(CC.PixelIdxList{ii}) = 0;
  end
end

figure()
imshow(BW)
Area=bwarea(BW);
CC=bwconncomp(BW);
if(Area>10000&& dist_b ~=0)
s=regionprops(CC,'Area','centroid','MajorAxisLength','MinorAxisLength','Orientation','Bounding
Box','EulerNumber','Perimeter');

Major=s.MajorAxisLength;
Minor=s.MinorAxisLength;
hold on
%plot(centroids(:,1),centroids(:,2),'b*');

%Bounding Box information, 1,2 left corner of box pixel, 3 is x length 4 is
%y length
p_A=[4695.43419202680,-40595.7005014340,98619.4503311143];
p_B1=[6.86887903612834,-75.6528307159344,278.225979474800];
p_B2=[13.1367701568270,-152.343060807504,623.315704575546];
dist_b=mean_dist(pp);
Area_r=p_A(1)*dist_b^2+p_A(2)*dist_b+p_A(3);
B1_r=p_B1(1)*dist_b^2+p_B1(2)*dist_b+p_B1(3);
```

```
B2_r=p_B2(1)*dist_b^2+p_B2(2)*dist_b+p_B2(3);
error=6000;
for k = 1 : length(s)
   AREA=s(k).Area;
   BB = s(k).BoundingBox;
   Orient=s(k).Orientation;
   centroids=cat(1,s(k).Centroid);
   holes=s(k).EulerNumber;
        if (abs(BB(3)-B1_r)<60 && abs(BB(4)-B2_r)<70 && abs(Orient)>75.0 &&
abs(AREA-Area_r)<error && abs(holes)<=10 && abs(cent(2)-240)<100)

       rectangle('Position', [BB(1),BB(2),BB(3),BB(4)],...
          'EdgeColor','r','LineWidth',2 )

       plot(centroids(1,1),centroids(1,2),'b*');
       centroid=centroids
       if exist('centroid')
          break;
       end
   else
       % centroids(k,:) = []

   end
end
else
   ang=0;
end
if exist('centroid')
   break;
end
end


% Calculating relative position using distance and centroid
% Image is a 480 x 640 480 tall, 640 wide
% knowing distance we should be able to calculate angle from center
% center is at 320 pixels
if exist ('centroid')
y_pixels=centroid(1)-320;
% conversion from pixels to meters,
pix_to_meters=0.006;

ang=atan2((y_pixels*pix_to_meters),dist_b);
goal_d_rel=dist_b;
goal_ang_rel=ang*-1;
else
   counter_k=counter_k+1;
   goal_d_rel=1.3;
   goal_ang_rel=0;
   ang=0;
end

angles=[angles,ang];
clear dist_b angle centroid
pause(1)
close all

end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. MOBILE ROBOT CALLBACK FUNCTION

A. MOBILE ROBOT CALLBACK USING OPENNI_TRACKER

```
function pwaypointCallback(~,message,cmdPub,cmdMsg)


% Control for Pioneer to drive  to goal position collected from a tf message


global derror
global goal
global cmd_vel_past
global cmd_vel
global goals
global odom
global counter_skeleton
global goal_local
global counter_pose



% Extract Data

pos = message.Pose.Pose;

quat = message.Pose.Pose.Orientation;

% Convert Quaternion to Euler Angles

angles = quat2eul([quat.W quat.X quat.Y quat.Z]);

theta = angles(1);
% goal(1)=goal(1)-0.25*cos(theta);
% goal(2)=goal(2)-0.25*sin(theta);

% State Vector

pose = [pos.Position.X, pos.Position.Y, theta]';  % X, Y, Theta
odom=[odom,pose];
if counter_skeleton==0
goal_local=goal+pose(1:2);
end
counter_skeleton=counter_skeleton+1;
counter_pose=0;

% Compute Distance to Goal and angle
```

```
derror = norm(goal_local-pose(1:2));

theta_g = atan2(goal_local(2)-pose(2),goal_local(1)-pose(1));

% set the local goal and derror if no new skeleton message has come in after 25 pose data
if counter_skeleton>25
   goal_local=[1.65;0]+pose(1:2);
   derror=1.65;
   theta_g=0;
elseif counter_skeleton>15
      goal_local=[1.55;0]+pose(1:2);
   derror=1.55;
   theta_g=0;
end
%ang_error = theta_g - pose(3);
ang_error = theta_g;

% Testing just using the goal position
%derror=norm(goal);
%ang_error=atan2(goal(2),goal(1));

% Proportional Guidance

if ang_error > pi

   ang_error = ang_error - 2*pi;

elseif ang_error < -pi

   ang_error = ang_error + 2*pi;

else

   ang_error = ang_error;

end

Kp_ang = 0.4; % proportinal gain for Angular velocity
Kp_dist =0.4; % proportional gain for Linear Velocity

Vnom = 0.2; % Nominal velocity
Anom=0.2;

%command of 1.0 in terminal, sends 1000 to robot
 if abs(ang_error) > 0.2
       cmdMsg.Linear.X = 0;
   cmdMsg.Angular.Z = Kp_ang*ang_error;
   if abs(cmdMsg.Angular.Z)>Anom
      if (cmdMsg.Angular.Z)<0.0
      cmdMsg.Angular.Z=-Anom;
```

```
        elseif (cmdMsg.Angular.Z)>0.0
          cmdMsg.Angular.Z=Anom;
        end
    end

elseif(derror<1.7)
     cmdMsg.Linear.X = -0.05;
     cmdMsg.Angular.Z = 0;
     fprintf('!!!!');
elseif (derror<1.85 && derror>1.7)
     cmdMsg.Linear.X = 0;
     cmdMsg.Angular.Z = 0;
     fprintf('!!!!');
else

    cmdMsg.Linear.X = min(Kp_dist*derror,Vnom);
    cmdMsg.Angular.Z =0;
end
cmd_vel=[cmdMsg.Linear.X;cmdMsg.Angular.Z];

% Create filter
A=0.2;
Yout=A*cmd_vel+(1-A)*cmd_vel_past;

%fprintf('Current pose X=%4.2f, Y=%4.2f, goalX=%4.2f, Y=%4.2f,
error=%4.2f\n',pose(1),pose(2),goal_local(1),goal_local(2),derror);

% Send Message
cmdMsg.Linear.X=Yout(1);
cmdMsg.Angular.Z=Yout(2);
send(cmdPub,cmdMsg);
cmd_vel_past=Yout;
goals=[goals,goal_local];
```

B. MOBILE ROBOT CALLBACK USING GOAL POSITION FROM MICROSOFT KINECT RAW DEPTH DATA

```
function pwaypointkinectCallback(~,message,cmdPub,cmdMsg)


% Waypoint control for Pioneer using goal position from Kinect raw data


global derror
global goal
global cmd_vel_past
global cmd_vel
global goals
global odom
global goal_d_rel;
global goal_ang_rel;
global counter;
global counter_k;


% Extract Data

pos = message.Pose.Pose;

quat = message.Pose.Pose.Orientation;

% Convert Quaternion to Euler Angles

angles = quat2eul([quat.W quat.X quat.Y quat.Z]);

theta = angles(1);


% State Vector

pose = [pos.Position.X, pos.Position.Y, theta]';  % X, Y, Theta
odom=[odom,pose];

% Compute Distance to Goal

% Compute absolute position of target in world frame if new goal_ang_rel is
% received
if (counter==0)
theta_t=goal_ang_rel;

Rz_robot=[cos(theta) -sin(theta)
        sin(theta) cos(theta)];
Rz_target=[cos(theta_t) -sin(theta_t)
        sin(theta_t) cos(theta_t)];
Pose_rel=[goal_d_rel;0];
Position_rel=Rz_robot*Rz_target*Pose_rel;
goal=[pose(1);pose(2)]+Position_rel
end

% counter is set to zero in the Kinect raw data callback when a new target position
% is determined
counter=1;

% Compute Distance to Goal

derror = norm(goal-pose(1:2));

% Proportional Guidance
```

```
theta_g = atan2(goal(2)-pose(2),goal(1)-pose(1));

%
ang_error = theta_g - pose(3);


if ang_error > pi

    ang_error = ang_error - 2*pi;

elseif ang_error < -pi

    ang_error = ang_error + 2*pi;

else

    ang_error = ang_error;

end

Kp_ang = 0.4; % proportinal gain for Angular velocity
Kp_dist =0.4; % proportional gain for Linear Velocity

Vnom = 0.2; % Nominal velocity
Anom=0.2;

if abs(ang_error) > 0.12
         cmdMsg.Linear.X = 0;
    cmdMsg.Angular.Z = Kp_ang*ang_error;
    if abs(cmdMsg.Angular.Z)>Anom
        if (cmdMsg.Angular.Z)<0.0
        cmdMsg.Angular.Z=-Anom;
        elseif (cmdMsg.Angular.Z)>0.0
          cmdMsg.Angular.Z=Anom;
        end
    end

elseif(derror<1.7)

     cmdMsg.Linear.X = -0.05;
     cmdMsg.Angular.Z = 0;
     %fprintf('!!!!');
elseif (derror<1.85 && derror>1.7)

     cmdMsg.Linear.X = 0;
     cmdMsg.Angular.Z = 0;
     fprintf('!!!!');

else
```

```
    cmdMsg.Linear.X = min(Kp_dist*derror,Vnom);
    cmdMsg.Angular.Z =0;

end

% if have not gotten a new goal position in 20 depth frames send a zero
% velocitycommand
 if counter_k>20
    cmdMsg.Linear.X = 0;
    cmdMsg.Angular.Z =0;
    fprintf('No new goal in 20 depth image frames\n');
 end

cmd_vel=[cmdMsg.Linear.X;cmdMsg.Angular.Z];

% Create filter
A=0.2;
Yout=A*cmd_vel+(1-A)*cmd_vel_past;

% Send Message
cmdMsg.Linear.X=Yout(1);
cmdMsg.Angular.Z=Yout(2);
send(cmdPub,cmdMsg);
cmd_vel_past=Yout;
goals=[goals,goal];
```

APPENDIX F. EXPERIMENT SCRIPT

```
% Control script for the P3-DX using both Openni skeleton tracker and Kinect raw data
% tracker

rosinit('http://192.168.0.2:11311', ...
        'NodeHost', '192.168.0.2');
    %%

global derror; % This is the distance to the waypoint - calc. in callback

derror = 10; % Initially we'll make this large


cmdpub = rospublisher('/my_p3dx/cmd_vel',rostype.geometry_msgs_Twist);

cmdmsg = rosmessage(cmdpub);

global goals;
goals=[0;0];
global odom;
odom=[0;0;0];
global goal;
goal=[0;0];
global goal_local
goal_local=[0;0];
global angles;
angles=[0, 0,0];
global Translation;
Translation=[0,0,0];
global counter_pose;
counter_pose=0;
global delta_angle;
delta_angle=[0, 0,0];
global angle_out;
angle_out=[0, 0,0];
global cmd_vel_past;
cmd_vel_past=[0;0];
global cmd_vel;
cmd_vel=[0;0];
global counter_skeleton;
counter_skeleton=0;
global counter;
counter=0;
global counter_k;
counter_k=0;
global goal_d_rel;
goal_d_rel=0;
```

```
global goal_ang_rel;
goal_ang_rel=0;


pause(8)

% start skeleton tracker callback
tf = rossubscriber('/tf_skeleton',{@tfCallback,cmdpub,cmdmsg});
pause(2)
%wait for skeleton tracker callback to get first goal position
while(goal(1)==0)
   pause(1)
end

% once have a goal position begin subscribing to the pioneer position

odomsub = rossubscriber('/my_p3dx/pose',{@pwaypointCallback,cmdpub,cmdmsg});
count=0;
while(1)
   while(1)
      pause(1)
      if (counter_pose>5)
         cmdmsg.Linear.X=0;

         cmdmsg.Angular.Z=0;

         send(cmdpub,cmdmsg);
      end
      if counter_skeleton> 25
         break
      else
         if count>0
         clear odomsub2;
         clear k_raw;
         odomsub =
rossubscriber('/my_p3dx/pose',{@Copy_of_pwaypointCallback,cmdpub,cmdmsg});
         end
      end
   end
   clear odomsub;
   k_raw =
rossubscriber('/kinect/depth/image_raw',{@kinectdepthCallback_lin,cmdpub,cmdmsg});
   while(goal_d_rel==0 && goal_ang_rel==0)
      pause(1)
   end
   odomsub2 = rossubscriber('/my_p3dx/pose',{@pwaypointkinectCallback,cmdpub,cmdmsg});
   count=count+1;
end

%%
```

```
% at the end of experiment send a zero velocity command and shutdown the MATLAB
% Ros node
clear odomsub
cmdmsg.Linear.X=0;

cmdmsg.Angular.Z=0;
send(cmdpub,cmdmsg);
rosshutdown
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Boston Dynamics. (n.d.). LS3 - Legged Squad Support Systems. [Online]. Available: http://www.bostondynamics.com/robot_ls3.html. Accessed Mar. 5, 2017.

[2] H. H. Seck. (2015, Dec. 22). Marine Corps shelves futuristic robo-mule due to noise concerns. [Online]. Available: http://www.military.com/daily-news/2015/12/22/marine-corps-shelves-futuristic-robo-mule-due-to-noise-concerns.html

[3] J. Han, L. Shao, D. Xu, and J. Shotton, "Enhanced computer vision with Microsoft Kinect sensor: a review," *IEEE Transactions on Cybernetics*, vol. 43, no. 5, pp. 1318–1334, Oct. 2013.

[4] L. Xia, C. C. Chen, and J. K. Aggarwal, "Human detection using depth information by Kinect," in *CVPR Workshops*, Colorado Springs, CO, 2011, pp. 15–22.

[5] L. Spinello and K. O. Arras, "People detection in RGB-D data," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, CA, 2011, pp. 3838–3843.

[6] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, San Diego, 2005, vol 1, pp. 886–896.

[7] C. Rougier, E. Auvinet, J. Rousseau, M. Mignotte, and J. Meunier, "Fall detection from depth map video sequences," in *9th International Conference on Smart Homes and Health Telematics*, Montreal, Canada, 2011, pp 121–128.

[8] Z. Zhang, W. Liu, V. Metsis, and V. Athitsos, "A viewpoint-independent statistical method for fall detection," in *Proceedings of the 21st International Conference on Pattern Recognition*, Tsukuba, 2012, pp. 3626–3630.

[9] E. Babaians, N. Khazaee Korghond, A. Ahmadi, M. Karimi, and S. S. Ghidary, "Skeleton and visual tracking fusion for human following task of service robots," in *3rd RSI International Conference on Robotics and Mechatronics*, Tehran, 2015, pp. 761–766.

[10] Q. Ren, Q. Zhao, H. Qi, and L. Li, "Real-time target tracking system for person-following robot," in *35th Chinese Control Conference*, Chengdu, 2016, pp. 6160–6165.

[11] E. Machida, M. Cao, T. Murao, and H. Hashimoto, “Human motion tracking of mobile robot with Kinect 3D sensor,” in *Proceedings of SICE Annual Conference*, Akita, 2012, pp. 2207–2211.

[12] T. K. Calibo, “Obstacle detection and avoidance on a mobile robotic platform using active depth sensing,” M.S. thesis, Dept. Elect. Eng., Naval Postgraduate School, Monterey, CA, 2014. [Online]. Available: http://calhoun.nps.edu/handle/10945/42591

[13] J. S. Lum, “Utilizing Robot Operating System (ROS) in robot vision and control,” M.S. thesis, Dept. Elect. Eng., Naval Postgraduate School, Monterey, CA, 2015. [Online]. Available: calhoun.nps.edu/handle/10945/47300

[14] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*, Sebastopol, CA: O'Reilly Media, Inc., 2015.

[15] I. McMahon. (2016, May 23). Client libraries. [Online]. Available: http://wiki.ros.org/Client%20Libraries

[16] MathWorks. (n.d.). MATLAB product page. [Online]. Available: https://www.mathworks.com/products/matlab.html. Accessed Mar. 15, 2017.

[17] MathWorks. (n.d.). MATLAB help page.[Online]. Available: https://www.mathworks.com/help/. Accessed Mar. 15, 2017.

[18] P. Corke, “Integrating ROS and MATLAB [ROS Topics],” *IEEE Robotics & Automation Magazine*, 2015, vol. 22, no. 2, pp. 18–20.

[19] MathWorks (n.d.). PDF documentation for Robotics System Toolbox. [Online]. Available: https://www.mathworks.com/help/pdf_doc/robotics/index.html. Accessed Mar. 15, 2017.

[20] Pioneer robotics. (n.d.). [Online]. Available: http://www.mobilerobots.com/Libraries/Downloads/Pioneer3DX-P3DX-RevA.sflb.ashx. Accessed Mar. 15, 2017.

[21] ARIA. (n.d.). Omron Adept MobileRobots, LLC. [Online]. Available: http://robots.mobilerobots.com/wiki/ARIA. Accessed Mar. 15, 2017.

[22] Pioneer P3-DX. (n.d.). Omron Adept MobileRobots, LLC. [Online]. Available: http://www.mobilerobots.com/Mobile_Robots.aspx. Accessed Mar. 15, 2017.

[23] Pioneer P3-DX mobile robot. (n.d.). Generation Robots. [Online]. Available: https://static.generation-robots.com/6629-large_default/robot-mobile-pioneer-3-dx.jpg. Accessed Mar. 17, 2017.

[24] Z. Zhang, “Microsoft Kinect sensor and its effect,” *IEEE MultiMedia,* 2012, vol. 19, no. 2, pp. 4–10.

[25] “PrimeSense NITE Algorithms 1.5,” (n.d). PrimeSense Inc. [Online]. Available: http://www.openni.ru/wp-content/uploads/2013/02/NITE-Algorithms.pdf. Accessed Mar 17, 2017.

[26] Kinect for Windows sensor components and specifications. (n.d.). Microsoft. [Online]. Available: https://i-msdn.sec.s-msft.com/dynimg/IC584396.png. Accessed Mar. 17, 2017.

[27] M. A. Livingston, Z. A. Sebastian, Z. Ai, and J. Decker, “Performance measurements for the Microsoft Kinect skeleton,” in *IEEE Virtual Reality Workshops*, Costa Mesa, CA, 2012, pp. 119–120.

[28] SUMICOM S675G3. (n.d.). [Online]. Available: http://www.kingyoung.com.tw/S675G3.htm. Accessed Mar. 16, 2017.

[29] NETGEAR RP614 web safe router. (n.d.). Web Collage. [Online]. Available: https://smedia.webcollage.net/rwvfp/wc/live/16282083/module/netgear/www.netgear.com/images/enus_diagram_backdiagram_rp61418-5526.gif.w960.gif. Accessed Apr. 22, 2017.

[30] Kinect Sensor. (n.d.). Microsoft Corporation. [Online]. Available: https://msdn.microsoft.com/en-us/library/hh438998.aspx. Accessed Apr. 22, 2017.

[31] RP614v1 – 4 port cable or DSL router with 10/100 Mbps switch. (n.d.). NETGEAR. [Online]. Available: https://www.netgear.com/support/product/rp614v1?cid=wmt_netgear_organic#GettingStarted_CommonTopics. Accessed Apr. 22, 2017.

[32] WNR612. (n.d.). NETGEAR. [Online]. Available: https://www.netgear.com/support/product/WNR612.aspx?cid=wmt_netgear_organic. Accessed Apr. 22, 2017.

[33] Optiplex desktop computers. (n.d.). Dell. [Online]. Available: http://www.dell.com/us/business/p/optiplex-desktops. Accessed Apr. 22, 2017.

[34] BurningIsoHowto. (2015, Mar. 29). Ubuntu. [Online]. Available:

https://help.ubuntu.com/community/BurningIsoHowto

[35] Installation. (2016, Jan. 04). Ubuntu. [Online]. Available: https://help.ubuntu.com/community/Installation#Installation_without_a_CD

[36] Repositories/Ubuntu. (2016, Sept. 20). Ubuntu. [Online]. Available: https://help.ubuntu.com/community/Repositories/Ubuntu

[37] Ubuntu install of ROS Indigo. (2017, Apr. 14). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/indigo/Installation/Ubuntu

[38] Running ROS across multiple machines. (2017, Mar. 14). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/MultipleMachines

[39] ROS/NetworkSetup. (2016, Apr. 13). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/ROS/NetworkSetup

[40] Creating a workspace for catkin. (2015, May 20). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/catkin/Tutorials/create_a_workspace

[41] ROSARIA. (2017, Mar. 01). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/ROSARIA

[42] How to use ROSARIA. (2016, Dec. 19). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/ROSARIA/Tutorials/How%20to%20use%20ROSARIA

[43] P2os. (2013, June 07). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/p2os

[44] MobileRobots/amr-ros-config. (n.d.). GitHub, Inc. [Online]. Available: https://github.com/MobileRobots/amr-ros-config. Accessed Apr 10, 2017.

[45] URDF. (2014, Oct 12). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/urdf

[46] Openni_launch. (2013, Nov 14). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/openni_launch

[47] Openni_camera. (2016, Apr. 27). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/openni_camera

[48] Avin2/SensorKinect. (n.d.). GitHub, Inc. [Online]. Available: https://github.com/avin2/SensorKinect. Accessed Apr. 10, 2017.

[49] Openni_tracker. (2013, Mar. 05). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/openni_tracker

[50] OpenNI SDK History. (n.d.). OpenNI. [Online]. Available: http://www.openni.ru/openni-sdk/openni-sdk-history-2/. Accessed Apr. 10, 2017.

[51] Hacking the Kinect 360. (2012, Aug. 15). [Online]. Available: http://1.bp.blogspot.com/-cRPcA5nbwgM/UBiC6YC0bFI/AAAAAAAAAqI/bQA5xr3AaqU/s1600/kinectpose-01.png

[52] Gmapping. (2015, Dec. 09). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/gmapping?distro=indigo

[53] Depthimage_to_laserscan. (2013, Mar. 10). Open Source Robotics Foundation. [Online]. Available: http://wiki.ros.org/depthimage_to_laserscan

[54] T. H. Dinh, M. T. Pham, M. D. Phung, D. M. Nguyen, V. M. Hoang, and Q. V. Tran, "Image segmentation based on histogram of depth and an application in driver distraction detection," in *13th International Conference on Control Automation Robotics & Vision*, Singapore, 2014, pp. 969–974.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California